FINAL REPORT

Providing Access to Graphical User Interfaces: The Mercator Project

July 25, 1996

Elizabeth Mynatt
GVU Center / College of Computing
Georgia Institute of Technology

TECHNICAL CONTACT: Craig E. Moore

SUMMARY: The purpose of this work was to enhance the existing Mercator research prototype to a commercial software system. The goal was to develop a commercial screen reader software system running on the Sun platform called UltraSonix. UltraSonix provides transparent access to X applications based on the Motif toolkit. Motif is the most common toolkit for building X applications; over 90 percent of X applications are built using Motif. UltraSonix allows transparent access of "off-the-shelf" Motif applications by blind users; no modifications to the applications are necessary to allow access.

Specific tasks included:

- Modifying Mercator to support Motif applications, including the Unix standard CDE environment on Sun SPARCstations.

  We have created a set of "rules" for translating Motif applications, as well as a set of "templates" that govern how the system responds to the individual Motif widgets. The system has been tested on the CDE environment running on Sun Solaris 2.4 and later.

  This is described in the UltraSonix User Manual, Section 6.2.1.

- Porting Mercator to the DEC Alpha platform

  This task comprised three activities: resolving 64-bit portability issues, AudioFile support, and resolving Solaris versus OSF/1 issues.

  This activity was contingent upon receiving a DEC Alpha workstation from Digital Equipment Corporation. DEC was unable to furnish us with a development workstation, so we were unable to resolve 64-bit or OSF/1 porting issues. We did, however, port the audio layer of the system to support DEC's AudioFile audio server.

  AudioFile support is described in the UltraSonix Design Document, Section 9.4.5.

- Expanding the I/O facilities to support voice input, Braille output as well as different speech synthesis systems.

  We have extended the I/O capabilities of the system to support multiple forms of speech, non-speech, and Braille I/O. The system currently supports the Dectalk DTC01, Dectalk Express, and TruVoice speech synthesizers, NetAudio and AudioFile audio servers, and the Alva line of Braille keyboards.

  I/O subsystem internals are described in the UltraSonix Design Document, Section 9.0; configuration of the I/O system is described in Section 9.3.4.

- Developing RAP (Remote Access Protocol) to support the use of the X11R6 standard "disability access hooks" by external agents such as screen readers. This effort was performed in conjunction with the X Consortium.

  The Remote Access Protocol work is ongoing within the X Consortium. Pieces of the proposal have been accepted as consortium standards however. The "ICE Rendezvous Mechanism," which specifies how UltraSonix will initially connect to client applications, has been accepted as a consortium standard and shipped in X11R6.1.

  The ICE Rendezvous Mechanism work was undertaken in conjunction with Digital Equipment Corporation, and is described in the UltraSonix Design Document, Section 4.3.

  The current proposed RAP spec is described in the UltraSonix Design Document, Section 4.2 and 4.5.

As we were unable to complete the port to the DEC Alpha workstation, we undertook several other tasks to improve the usability and configurability of the system.

Additional tasks completed:

- Console Support.

  We extended the system to support "console" applications that act as "controlling terminals" for the UltraSonix system. Two consoles ship with UltraSonix, a command-line version, and a fully graphical version. These consoles allow easy end-user customization and control of UltraSonix.

  The consoles are described in the UltraSonix Design Document, Section 10.2.

- Extensible Text Filters.

  UltraSonix now has the capability to support user-supplied "text filters" that control the output of text in on-screen text areas. These can be easily defined by end-users to customize the system.

  Text filters are described in the UltraSonix Design Document, Section 6.3.3

- Solaris Package Format Installation.

  The UltraSonix software for Solaris ships in the standard "package" format. This format allows easy installation and upgrade of the software, either from the command line or via the Software Manager GUI tool that ships with Solaris.

  Installation of package files is described in the UltraSonix User Manual, Section 2.2.

BACKGROUND: This project was undertaken to provide transparent access to X Windows application for people who are blind or severely visually impaired. This goal requires that the interactive visual interface be transformed into an interactive nonvisual interface. Software that performs this type of task is typically called a screen reader. Much of the work in developing screen readers has been fueled by federal legislation requiring access to electronic equipment.

With the introduction of graphical user interfaces, the difficulty of providing access to visual interfaces increased dramatically. This increase was due to the use of bit-mapped displays. The screen reader can no longer utilize the framebuffer to determine the contents of the screen, but must somehow trap the drawing requests originating from the graphical applications. Also, the use of graphical objects or icons as well as the use of the mouse, make the job of transforming a graphical interface into another modality extremely difficult. At this time access solutions for the Macintosh and Microsoft Windows are available commercially. Only X Windows is still inaccessible, although, this project has developed a screen reader for X Windows which only needs further commercialization work to turn it into a commercial product.

APPROACH: Three major prototype systems were developed to investigate software architectures for supporting transforming graphical interfaces into nonvisual interfaces. The final prototype utilizes three extensions to X Windows to support screen reader access. Two of these extensions were developed by this project. The first extension, called the Xt-based protocol, provides asynchronous communication between the X application and the screen reader. This protocol can be used to inform the screen reader when objects such as windows are created, when they are modified, and when they are deleted (or unmapped) such as a dialog box disappearing from the screen. The second extension, called the Xlib hook, is used to trap all information that bypasses the Xt hooks. This information is typically low-level information such as simple drawing and text rendering requests. These two hooks used together provide sufficient information to model a graphical application. The last extension, called XTest, not developed by this project, is used to simulate mouse input by a blind person using either keyboard or voice input. The information about the X application is stored in an off-screen model. The data is stored according to the widget hierarchy which was used to create the X application. This tree structure represents parent-child relationships between the interface objects. Resources (or attributes) of the interface objects are also stored. The screen reader (called Mercator) then provides translation rules for the different types of interface objects. These rules specify how different interface objects, such as a push button or menu, are represented and how they respond to user input.

ACCOMPLISHMENTS: The accomplishments of the Mercator project have been two-fold. First, in order to provide transparent access to X applications, a software framework (or environment) was developed which can monitor (watch for changes in the state of application), model (build a useful off-screen model of the application interface) and translate (provide "rules" for translating the graphical interface into a nonvisual interface) X applications as well as providing new ways to send user input to the application. Second, this project has designed and informally evaluated a "hear-and-feel" methodology for transforming graphical interfaces into nonvisual interfaces. This task is important because little is known about effective ways to effectively represent graphical interfaces for blind computer users. In summary, this project has demonstrated the feasibility of providing access to X Windows for blind computer users. At this time, the researchers at Tech are the only ones world-wide to build such a system. Tech has acted as champion and designer of the disability access hooks for X Windows. These hooks are now part of the general X Windows distribution. They support the use of screen readers as well as other applications which need to monitor and configure the execution of a graphical X applications. Through publishing, demonstration and presentations, Tech has successfully introduced new interaction techniques for screen readers. These techniques, such as the use of auditory icons and hierarchical navigation, provide an intuitive and efficient interface to graphical applications for blind computer users. This project establishes Georgia Tech as the primary researchers in this area. This recognition allowed them to begin working with the newly formed DACX committee and the X Consortium on modification to X Windows to support access for people with disabilities. The DACX (Disability Action Committee on X) is a national, vendor-neutral committee which is helping design and implement standard access solutions for people who want to use X Windows.

This commercialization effort is jointly supported by Georgia's Advanced Development Technology Center's Faculty Commercialization Grant.

PUBLICATIONS AND PATENT APPLICATIONS:

Edwards, W.K., Liebeskind, S. H., Mynatt, E.D and Walker, W.D. A Remote Access Protocol for the X Window System. In the Proceedings of the 9th Annual X Technical Conference, Boston, MA, 1995.

Edwards, W.K. and Mynatt, E.D. An Architecture for Transforming Graphical Interfaces. In the Proceedings of UIST'94: User Interface Software and Technology Symposium, Marina Del Ray, CA., Nov. 2-4, 1994, 39-47.

Edwards, W.K., Mynatt, E.D. and Stockton, K., "Providing Access to Graphical User Interfaces - Not Graphical Screens," Proceedings of ASSETS `94, November 1994.

Edwards, W. K. and Rodriguez, T. Runtime Translation of X Interfaces to Support Visually-Impaired Users. In Proceedings of the 7th Annual X Technical Conference, Boston, MA, 1993.

Johnson, E., Mynatt, E.D., Novak, M., and Walker, W., "Extending the User Interface for X Windows to Include Persons with Physical and Sensory Disabilities: The DACX Project," in the Proceedings of the Closing the Gap Conference, Minneapolis, MN, October 1993.

Mynatt, E.D., "Transforming Graphical Interfaces into Auditory Interfaces for Blind Users," to appear in ACM Transactions on Computer-Human Interaction, ACM, 1997.

Mynatt, E.D. and Edwards, W.K., "Metaphors for Nonvisual Computing," Extraordinary Human-Computer Interaction, (editor) Dr. Alistair Edwards, University of York, Cambridge University Press, 1996.

Mynatt, E. Transforming Graphical Interfaces into Auditory Interfaces. Doctoral Dissertation, Georgia Institute of Technology, Atlanta. 1995.

Mynatt, E. Designing Auditory Icons, In Proceedings of the Second International Conference of Auditory Display, ICAD `94, Sante Fe, New Mexico, 1995, pp. 109-120.

Mynatt, E.D., "Auditory Presentation of Graphical User Interfaces," Auditory Display: Sonification, Audification and Auditory Interfaces, Ed. G. Kramer, SFI Studies in the Sciences of Complexity, Vol 18, Addison Wesley, 1994.

Mynatt, E.D. and Weber, G., "Nonvisual Presentation of Graphical User Interfaces: Contrasting Two Approaches," in the Proceedings of the 1994 ACM Conference on Human Factors in Computing Systems (CHI'94).

Edwards, A., Edward, A.D.N. and Mynatt E.D., "Enabling Technology for Users with Special Needs", in the Proceedings of INTERCHI'93, 1993 Conference on Human Factors in Computing Systems and in the Proceedings of the 1994 ACM Conference on Human Factors in Computing Systems (CHI'94). and in the Proceedings of the 1995 ACM Conference on Human Factors in Computing Systems (CHI'95).

ATTACHMENTS:

Transforming Graphical Interfaces into Auditory Interfaces for Blind Users (to appear in ACM Transactions on Computer-Human Interaction, ACM, 1997).

UltraSonix Design Document.

UltraSonix User Manual.

Transforming Graphical Interfaces into Auditory Interfaces for Blind Users

Elizabeth D. Mynatt[1]
Xerox Palo Alto Research Center
mynatt@parc.xerox.com

ABSTRACT

While graphical interfaces have provided a host of advantages to the majority of computer users, they have created a significant barrier to blind computer users. To meet the needs of these users, a methodology for transforming graphical interfaces into nonvisual interfaces has been developed. In this design, the salient components of graphical interfaces are transformed into auditory interfaces. Based on a hierarchical model of the graphical interface, the auditory interface utilizes auditory icons to convey interface objects. Users navigate the interface by traversing its hierarchical structure. This design results in a usable interface that meets the needs of blind users while providing many of the benefits of graphical interfaces.

KEYWORDS auditory interfaces, auditory icons, blind users, assistive technology, UI models

INTRODUCTION

The problem addressed by this research can be simply stated, "What kind of interface would you design for a blind person using a graphical user interface?" The requirements of blind users demand a *general* mechanism for *transforming* graphical interfaces into nonvisual interfaces. Additionally, blind users would like to enjoy the benefits of graphical user interfaces. The requirements of blind users coupled with the benefits of graphical interfaces form a set of goals for the interface transformations. We will see that commercial software for blind users fails to meet many of these goals. Specifically, the reliance on a spatial model of the graphical interface impacts the usability of the resulting nonvisual interface.

The first step in transforming graphical interfaces into nonvisual interfaces is determining the contents of the transformation. What is being converted from the graphical modality into the nonvisual modality? Graphical interfaces are composed of groups of interface objects that are presented spatially on a two dimensional display. These objects are characterized by a number of attributes that help convey their intended functionality. The objects making up the graphical interface, their attributes and the relationships between the objects comprise the contents of the interface transformation.

Next, it is necessary to model the contents of the transformation so that the model both captures the critical characteristics of the graphical interfaces and provides the basis for an intuitive nonvisual interface. Different models are possible. After comparing spatial, hierarchical and conversational models, we argue for

1. This work was conducted when the author was at the Georgia Institute of Technology.

utilizing a hierarchical model because it best captures the underlying structure of the graphical interface without requiring application, domain-specific knowledge.

Given a hierarchical model of the graphical interface, the next step is designing the nonvisual interface. In this design, we have focused on conveying the contents of the user interface, supporting navigation, and providing controls for manipulating the interface. By using auditory cues akin to sound effects heard in real world environments, we attempt to provide the benefits of iconic representations. These auditory icons[14] convey the type of interface objects as well as attributes of the objects such as their size, selection state, and spatial location. For example, a muffled, light switch sound conveys a greyed-out toggle button.

Users move from object to object based on the hierarchical model of the interface. Interruptability, navigation shortcuts, and previews help alleviate the potential tedium of traversing a large structure. Auditory feedback also helps users perceive changes in the interface based on their input or application events. For example, rising and falling whistling sounds accompany the appearance and disappearance of pop-up windows.

Assessments of this design are based on over four years of feedback from blind computer users as well as controlled experiments with sighted users. We conclude this paper by evaluating this design against the goals that we outlined for interface transformations. Although lacking in its ability to present information spatially, this design results in a usable interface that provides many of the critical characteristics of graphical interfaces.

BACKGROUND

The GUI Problem

The 1990 paper "The Graphical User Interface: Crisis, Danger and Opportunity" [5] summarized an overwhelming concern expressed by the blind community: a new type of visual interface threatened to erase the progress made by the innovators of screen reader software. Such software (as the name implies) could read the contents of a computer screen, allowing blind computer users equal access to the tools used by their sighted colleagues. Whereas character-based screens were easily accessible, new graphical interfaces presented a host of technological challenges. The contents of the screen were mere pixel values, the on or off "dots" which form the basis of any bit-mapped display. The goal for screen reader providers was to develop new methods for bringing the meaning of these picture-based interfaces to users who could not see them.

The crisis was imminent. Graphical user interfaces were quickly adopted by the sighted community as a more intuitive interface. Ironically, these interfaces were deemed more accessible by the sighted population because they seemed approachable for novice computer users. The danger was tangible in the forms of lost jobs, barriers to education, and the simple frustration of being left behind by the computer industry.

Much has changed since that article was published. Commercial screen reader interfaces now exist for two of the three main graphical environments. But many blind users still do not view graphical interfaces as a new opportunity. Screen readers designers, faced with the task of translating a complex, visual interface into auditory or tactile output, have attempted to create one-to-one translations of the spatially arranged graphical interfaces. Blind users have responded with difficulties in using these visually-oriented interfaces.

The Mercator Project at Georgia Tech addressed two untouched areas of work in the screen reader community. First, no one had designed a screen reader for X Window applications, such as Motif applications used in research, business and educational settings. Second, there was little work in alternate representations of graphical interfaces that were not based on speech output and spatial organizations.

The implementation of Mercator is described in [11][12][13]. Briefly, the system provides the infrastructure to monitor and model unmodified X applications while they are running. A collection of "hooks"[1] in the Xlib and Xt Intrinsics libraries of the X Window System trap interesting events such as the creation of a push button or the appearance of a window. The information gleaned from these hooks is transmitted to Mercator

which creates a model of the graphical interface based on the application's widget hierarchy. Mercator also provides facilities for creating interfaces to replace or augment the graphical interface. Interface behavior can be specified in an interpreted language supporting prototyping and end-user customization.

An underlying assumption in the design of Mercator interfaces is the dominant use of auditory output. Researchers have experienced limited success with tactile devices with the exception of braille output. Additionally, a significant portion of people who are blind also suffer from diabetes which reduces their sensitivity to tactile stimuli [17]. Nevertheless Mercator includes a tactile component as well. For example, since speech synthesizers are notoriously bad at reading source code, Mercator provides a Braille terminal as an alternate means for presenting textual information

Requirements for GUI Access by Blind Users

The requirements for the auditory interface are driven by the need for blind users to work with their sighted colleagues employing the same graphical applications. Without knowledge of the application domain, the screen reader system must transform the contents of the graphical interface into a usable auditory interface. By monitoring the execution of a graphical application, the screen reader creates a model of the application interface and derives a complimentary auditory interface. The user's interaction with the auditory interface is forwarded to the graphical interface and the process continues as shown in the following figure.

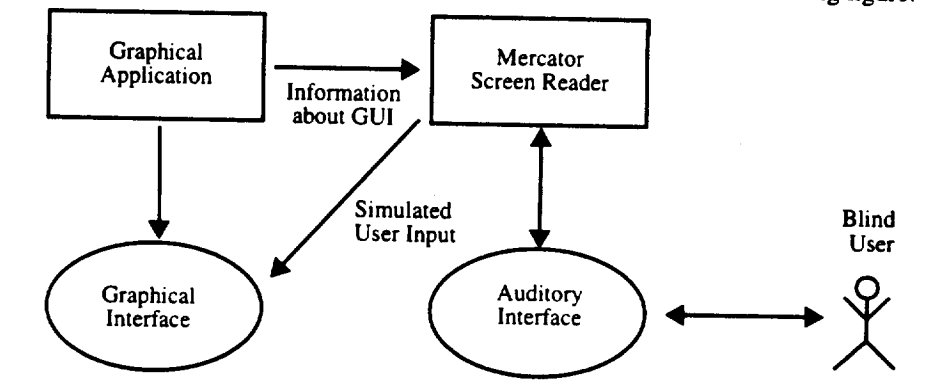

FIGURE 1: Simplified View of GUI to AUI Transformation

The foremost critical requirement is *transparent transformations* of graphical interfaces. Modifying individual applications does not address problems of providing access to a set of graphical applications. A general mechanism for transforming any X Window application is needed. The ideal scenario is that blind users running the screen reader on their systems should be able to use any X application without needing to specially tailor the application interfaces.

To address this need, Mercator automatically transforms text-based, X Windows applications while they are running, providing an auditory interface. This requirement for transparent transformations impacts the design in two critical ways. First, there is no domain knowledge to inform the creation of the auditory interface. Mercator is unaware of the functionality of the application, such as whether it is a word processor or electronic mail tool, but is only aware of how the graphical interface is constructed. Second, this transformation is done in "real-time". There is no off-line processing or analysis of the graphical interface.

An implicit requirement for screen reader systems is that they *facilitate collaboration between sighted and blind colleagues*. Blind users do not work in isolation from their sighted counterparts. Therefore it is imperative that blind and sighted users be able to communicate about their use of application interfaces.

---

1. The "hooks" are now part of the standard X Windows System since X11R6. The protocol used to transmit information trapped by the hooks to an external program is under consideration by the X Consortium.

In addition to reinforcing the need for transparent access, this requirement constrains the design of the auditory interface. While an auditory interface to an application may be quite intuitive and usable, if it does not express interface concepts similar to the graphical interface, it does not solve the collaboration need by the blind computer user. Ideally a blind user should be able to ask a sighted user how to do something with an application interface and be able to utilize directions expressed in terms of the graphical interface. This ideal scenario is difficult to achieve, but the design goals of designing for collaboration versus designing for intuitive auditory interaction conflict in interesting ways.

Within the range of graphical interfaces, this work focuses on the transformation of text-oriented graphical interfaces such as electronic mail programs, word processors and spreadsheets. By choosing this area, we are focusing on interfaces in which text is the primary object of interest and where text is manipulated through the use of graphical controls. In contrast, applications such as drawing programs where graphics are the primary objects of interest are not addressed in this work. This restriction is due to the additional difficulty of representing pure graphical information in the auditory modality. Nevertheless the chosen application set is sufficiently interesting since it represents applications that are commonly used.

Expressed as a general requirement, an additional need by blind computer users is to *experience the benefits of graphical interfaces* enjoyed by their sighted counterparts, such as iconic representation and direct manipulation. Below, we present goals for screen reader interface design based on the benefits of GUIs:

- **Access to functionality**

  At a minimum, the user must be able to use the functions represented by the graphical interface. For example, in a word processor where pull-down menus support operations for loading and saving files, users would need an interface to this functionality. Some software vendors maintain that their graphical applications are accessible to blind users because they provide a separate command-line interface that can be read by older screen readers. Simply providing access to the same functionality likely breaks the goal of supporting collaboration between blind and sighted users since they use a distinctly different interface.

- **Iconic representations of interface objects**

  Graphical icons, from trashcans to push buttons, help the user assess the capabilities of an interface by leveraging knowledge of the physical world. Visual attributes of interface objects such as size and highlighting also convey information to the user.

- **Direct manipulation**

  Closely coupled with the benefit of iconic representation, is the benefit of direct manipulation. This benefit is achieved when the user is able to directly interact with objects of interest to the task at hand, and output in the interface is expressed via these objects[18].

- **Spatial arrangement**

  Graphical interfaces allow the user to organize information in a 2 1/2D space. Contrast organizing a desktop by maintaining lists of objects and categories of lists. Another benefit of spatial arrangement is that it can leverage knowledge of the physical world. Sliders that support viewing portions of a document capitalize on moving sheets of paper sideways and front-to-back in a stack.

- **Constant presentation**

  A benefit of visual interfaces is that they exist in physical space that can be reviewed over time. This advantage of the visual sensory system is capitalized in graphical interfaces. These displays serve as a surrogate short-term memory for recalling the contents of the user interfaces.

We will see that these benefits are ordered from easiest to hardest for a screen reader system to provide. In the following section, we briefly evaluate screen reader systems that allow blind users to interact with representations of graphical interfaces.

Evaluation of Screen Reader Interfaces

There are two general classes of commercial screen readers that provide auditory interfaces for graphical interfaces. The first class is dominated by a product called OutSpoken [1]. The primary characteristic of this class is that the structure of the auditory interface is based primarily on the spatial layout of the graphical interface. Users navigate the screen using the mouse or keyboard shortcuts. The interface uses synthesized speech almost exclusively. At the basic level, the user moves the mouse cursor across the screen, and when the cursor intersects a graphical object the speech synthesizer reads information about that object. An auditory cue is used to convey moving across a window boundary.

Since OutSpoken relies heavily on optical character recognition (OCR) algorithms that are extended to recognize graphical icons, this interface does not group icons as one might expect. Two examples of OutSpoken's interface reveal its usability limitations.

In a grouping of controls, such as these in the following figure, the users must access the controls in terms of their visual layout. For example to move from "Row" to "Selection", the user must move down twice. There is little information conveyed by this spatial layout, but the arrangement was chosen because it fit well within the dialog box. The user could as easily move to the right twice. This interaction style requires blind users to memorize visual layouts that conveys little meaning about the interface.

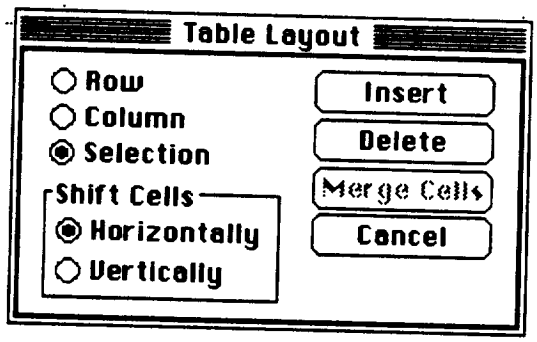

FIGURE 2: OutSpoken Example of Row-Column Navigation

The conceptual model that underlies the OutSpoken interface is the arrangement of information in a row-column format. This model was chosen because it is similar to previous text-oriented screen reader interfaces. Because the OutSpoken interface imposes little hierarchy (windows are the only grouping mechanism), moving through the objects in the above dialog would result in this order of spoken output: Row, Insert, Column, Delete, Selection and so on. Users are confused by this interaction since the semantic groupings that are obvious in the visual interface are not conveyed in the auditory interface

ScreenReader II by IBM, WindowBridge by SynthAVoice, and ProTalk by Hinter Joyce are products that provide access to the Microsoft Windows environment. As representatives of the second class of screen readers, these interfaces require the use of existing keyboard shortcuts provided by the Windows environment. Like OutSpoken, these products use only synthesized speech and braille output.

The reliance on the Windows keyboard shortcuts creates most of the usability problem with these screen readers. First, while the shortcuts provide more structural information than OutSpoken, they are designed to be augmented with the information in the visual display.

A more troublesome problem is that the shortcuts only allow the user to navigate to graphical objects that accept user input. Areas such as greyed out buttons and message bars are "invisible." In order to access read-only information, the user must define view areas by a row-column position per application. The user can then create keyboard macros to read the information at a particular view area. These view areas are defined in a separate file or application profile, and this process requires the assistance of a sighted person.

MODELING GRAPHICAL INTERFACES

Determining the Contents of the Interface Transformation

To motivate creating a model of a graphical interface, we examine a typical graphical interface as shown in the following figure. The question to be answered at this stage is:

> *"What are the characteristics and components of this interface that are critical to its use?"*

In contrast to the previous discussion on the advantages of graphical interfaces, during this section we need to categorize information about graphical interfaces that will be stored in our model.

What are the objects?

A fundamental notion behind graphical interfaces is that the user directly interacts with *things*: objects or interactors that can be manipulated by the user in a set number of ways. In the example, there are a number of objects such as windows, radio buttons, push buttons, scroll bars, editable text areas, and read-only text areas such as message bars. These objects form the basis for how we conceptualize a graphical interface.

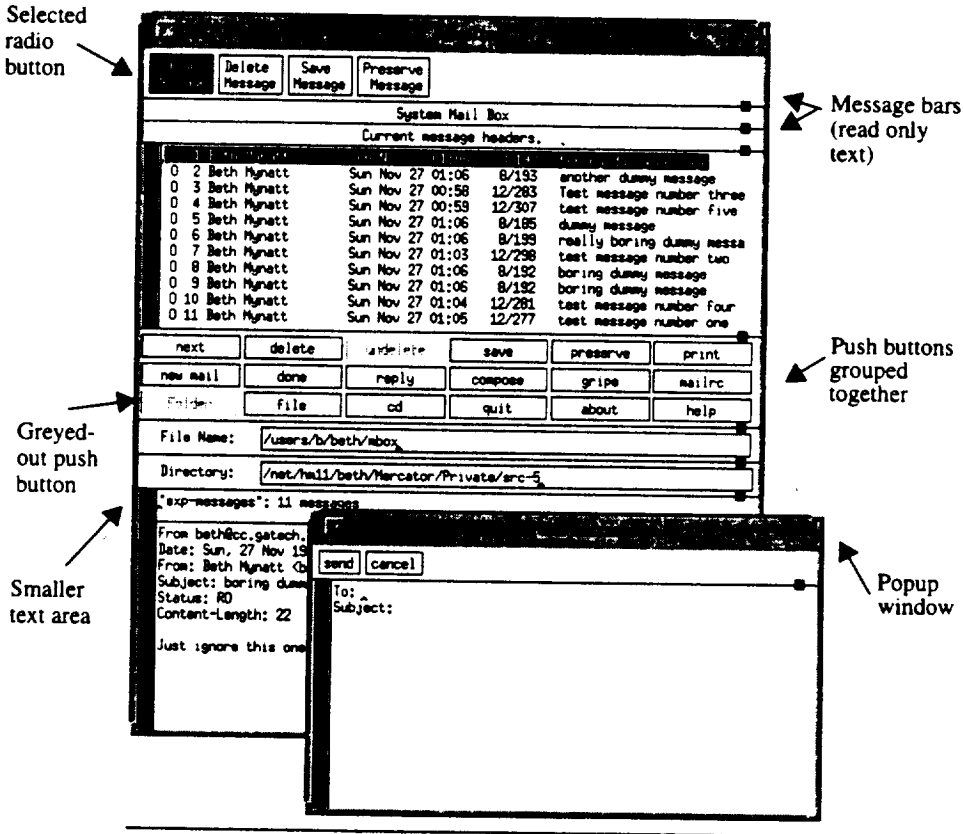

FIGURE 3: Annotated Graphical Interface

What are attributes of the objects?

Most interface objects are characterized by a number of visual attributes that help clarify their use. For example, many interactors can be highlighted (often indicating a current selection) and greyed out (indicating that the object is unavailable for use).

The relative sizes of different objects may be informative. In the screenshot, one of the text areas is much smaller than the others. This size indicates the type of text presented in this space, in this case, short diagnostic messages. Other size attributes are related to collections of objects. For example, the four radio

buttons take up less space than the eighteen push buttons in the sample interface. Because sighted users quickly scan this information, they internalize that the radio buttons represent a smaller set of choices.

Other attributes are related to the spatial distribution of objects. For example, the sample interface is meant to be *read* from top to bottom, left to right, following Western reading conventions.

What are the affordances of the objects?

Objects in graphical interfaces can be categorized by their basic functionality. Many objects provide a means to group other objects. In this example, both windows and boxes collect objects into meaningful groups.

Users interact with objects in graphical interfaces in a set of predetermined ways. For example, a host of button objects support different forms of selection.

Text objects support the entering and manipulation of textual information, although the behavior of a text object can be constrained to indicate information about its contents. For example, the text in the first, large text object of the pictured interface, supports selecting a line of text since those lines represent email message headers. Some text objects support reading, but not editing of their contents.

What are the relationships between objects?

Another important question is how does the user perceive that objects are related to each other. Commonly, relationships are expressed via grouping. For example, the four radio buttons are not randomly scattered throughout the interface but are grouped together. In general, hierarchical relationships among objects inform us about the structure of the application interface. As already discussed, windows, boxes and white space convey groups of related objects.

Another primary relationship is cause and effect. In the example, selecting the push button "reply" causes the dialog box to be popped up. Selecting a message header causes the message to be displayed. If the interface response time is short, the user will *associate* these objects as having a cause-and-effect relationship.

What are the names of objects?

Although not visually depicted, many objects in the graphical interface have common names associated with them such as window, push button, and scroll bar. Since sighted and blind users will need to communicate with each other about application interfaces, it is necessary to retain naming conventions.

Completeness?

Is our model of the graphical interface complete? Depending on how we represent the model of the graphical interface with auditory cues, it may appear that we are discarding information in the graphical presentation. For example, we may not attempt to present objects at specific x,y location, but we may use the x,y coordinates to help determine the order of objects in the auditory interface. Likewise, we may not convey the amount of overlap between partially occluded windows, but we will likely support the notion of *focus* in the auditory interface.

There will likely be information in the graphical interface not conveyed in the auditory interface such as line width or color when these attributes do not convey information. The difficulty is determining, given a generic transformation, when the visual attribute is not meaningful.

Additionally, there are characteristics of the graphical interface that are difficult, if not impossible, to convey in an auditory interface, such as a persistent overview of the interface. How information about the graphical interface is utilized is determined by the underlying representation of the interface. In the next section, we compare three potential classes of models that could be used to represent the information about the graphical interface. This model provides the basis for the auditory interface.

Modeling the GUI

The next step in the design process is determining a model for graphical user interfaces. Since the model impacts both the user interface as well as the system design and implementation, it is necessary for us to consider the following questions when evaluating possible models:

- How well does this model capture important GUI characteristics?
- What kinds of auditory interfaces could be based on this model?

As an extreme example of a possible model, we could attempt to represent the GUI interface with musical notation. Although it would be easy to create an auditory interface based on musical notation, it would be quite difficult, if not impossible, to represent GUI characteristics with musical notation.

During this discussion, we compare three types of potential conceptual models. These are:

- **Spatial Models**

  The graphical interface is modeled as a 2 1/2 dimensional projection in space easily capturing aspects of the GUI such as the spatial distribution of objects. This model is primarily used by commercial screen readers.

- **Hierarchical Models**

  The graphical interface is modeled as a hierarchical structure, such as a tree or outline easily capturing parent-child grouping relationships. Most phone-based auditory interfaces utilize hierarchical interfaces implemented with menus.

- **Conversational Models**

  The graphical interface is modeled as a dialogue where the user can converse with the auditory interfaces. Natural language understanding coupled with voice recognition systems are used to implement these interfaces.

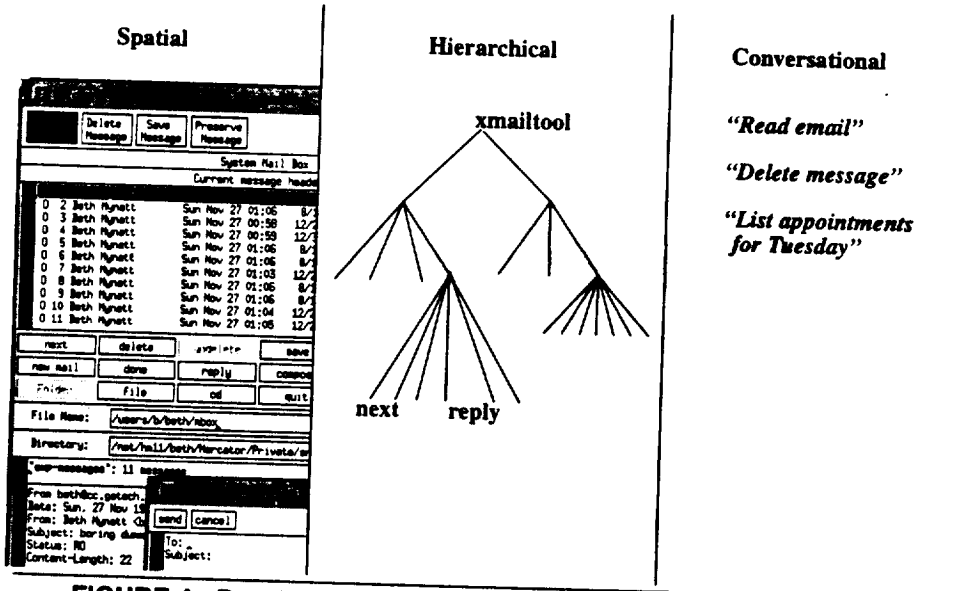

FIGURE 4: Possible Conceptual Models for Graphical Interfaces

Assessing Spatial Models

Using spatial models to represent graphical interfaces is attractive since graphical interfaces are presented using a spatial metaphor. The sighted user is presented with a spatially arranged picture of interface objects that can stack on top of each other in a 2 1/2D fashion. Many of the advantages of graphical interfaces,

discussed previously stem from their static, spatial presentation. Obviously representing the graphical interface with a spatial model is not difficult, so the remaining question is what type of auditory interface could be based on this model.

The major difficulty with spatial models is that auditory interfaces are limited in their ability to present information spatially. Since 3D spatial sound systems cannot be used to produce a one-to-one mapping of the visual space to an auditory space, it would be necessary to present an abstraction of the graphical display. Just as maps serve as abstractions for physical space, the goal would be to create an auditory abstraction for the graphical space. Although this approach is feasible and worth future investigation, the analysis of existing screen readers points to two problems with this approach.

First, as discussed during the review of screen reader interfaces, blind users find it difficult to work with spatially arranged user interfaces. Many users conceptualize the interface based on its logical structure and then attempt to memorize the spatial presentation. Although it is clear that blind people can successfully navigate physical spaces such as their home, one user likened working with a graphical interface with trying to navigate a large, unknown room where it is "easy to get lost and become disoriented [7]."

Second, although one could argue that existing screen reader interfaces have not provided the right spatial abstraction for a graphical interface, finding such an abstraction is difficult because graphical interfaces have been optimized for visual presentation. The need to fit the graphical interface into a limited visual space results in spatial layouts that are not informative, but are the result of conserving screen real estate. In a generic analysis of an X Windows graphical interface, it is impossible to determine *when* spatial layouts are informative. Although current screen reader interfaces have attempted to provide the benefits of a spatial organization, their users more often are confused by spatial arrangements that convey no meaning.

Assessing Hierarchical Models

Many auditory interfaces are based on hierarchical models[21][26]. For example, interfaces for voice mail allow the user to navigate through a hierarchy of choices for listening to and deleting messages. Hierarchical models are used because they can abstractly represent groups. It is also relatively easy to navigate these auditory interfaces using keypad or voice input, although the requisite path from one object to another may be lengthy. Since hierarchical structures represent discrete, as opposed to continuous, values, they are well suited for conveying discrete objects.

Given that there are previous examples of complex, hierarchical auditory interfaces, the primary question is how well graphical interfaces can be modeled using a hierarchical structure. A tree-structure representation of the graphical interface in Figure 3: "Annotated Graphical Interface" is shown in the following figure. The tree structure lends itself to representing the objects in their interface, as well as the parent-child relationships between those objects. Cause-and-effect relationships can be modeled as additional links in the structure. In the example, pushing the reply button causes the pop-up dialogue to appear.

A significant limitation of this model is that it does not capture visual attributes of the graphical interface. Some representations of visual cues are possible. For instance, the ordering of objects in the structure can be based on their spatial arrangement in the graphical interface. Likewise the size of grouping objects, such as windows, is partially represented as the number of children. Nevertheless, this model suffers from its inability to explicitly represent all the visual characteristics of the graphical interface, These characteristics can be stored as attributes of the objects in the hierarchical structure. The auditory interface would be responsible for conveying these attributes, in addition to conveying the underlying model.

Assessing Conversational Models

Another class of auditory interfaces commonly uses conversational dialogues as the basis for the user interfaces[21][26]. For example, both Stifelman's Conversational VoiceNotes [28] and Yankelovich's SpeechActs[29][30] utilize voice recognition technology as the primary means of input to an auditory interface. VoiceNotes provides an interface to a hand-held notes organizer while SpeechActs provides an interface to desktop applications such as email and calendar. Typical user input phrases are:

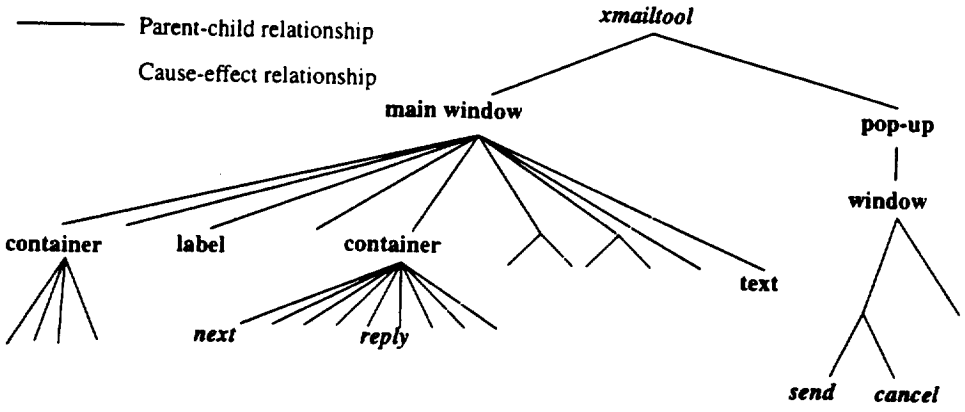

FIGURE 5: Partial Tree Structure Representation of Graphical Interface

List notes for July 12th

New appointment with Jim Foley this Friday at 3pm

Both of these interfaces are replicating functionality that can be found in a graphical interface. SpeechActs is actually a front-end for graphical desktop programs. Given that it is possible to create useful auditory interfaces using conversational models, the remaining question is how does this model work with our goal of modeling graphical user interfaces. There are two problems with using these models for our task:

- Requires Domain Knowledge

  The example input phrases above illustrate that these interfaces rely on understanding the domain of the application interfaces. In our automatic analysis of graphical interfaces, it is unlikely that we will obtain sufficient information to build a domain-dependent dialogue.

- Interaction Significantly Different than Graphical Interface

  Sighted users and blind users will not have the same building blocks for discussing how to operate an interface since the conversational interface hides components of the GUI such as menus and buttons.

Choosing a Model

We have based our representation of the graphical interface on a **hierarchical** conceptual model since best captures the underlying structure of the graphical interface without requiring domain-specific knowledge of the graphical application. The primary relationship represented in the hierarchical model is the parent-child relationship between interface objects. These relationships appear to be the basis for how blind users conceptualize graphical interfaces. In many ways, they are likely to be the basis for how all users conceptualize graphical interfaces given the importance of structural information in this class of interfaces. Spatial organizations are problematic since graphical interfaces typically generate spatial layouts based on space-conserving constraints that are often confusing for blind users. Conversational models require domain knowledge to capture the functionality specific to the application interface. This information would be extremely difficult, if not impossible, to obtain from a generic X Windows application.

One important limitation of the hierarchical model is that it does not effectively capture the power of a visual, spatial presentation. Two advantages of the visual interface are that the user can quickly recognize interface objects from the bit mapped pictures on the screen, and that the user can quickly scan the collection of onscreen objects. Therefore, two critical requirements of the design are that the user can quickly recognize interface objects and that the user can quickly survey the contents of the interface.

MERCATOR INTERFACE DESIGN

In this section, we describe the basic interface design for Mercator. The primary question that we address is:

> *Given the hierarchical model of the graphical interface, what auditory interface do we present for a blind user?*

The inherent disadvantage of all auditory interfaces is they are largely invisible. For this reason, a significant portion of this design will focus on **conveying the contents** of the auditory interface. Users must be able to determine the identity and attributes of the various objects that make up the auditory interface.

In addition to recognizing individual objects, the user must be able to navigate the space of the interface. The controls for **navigation** must support the user's mental model of the auditory interface. For example, moving the mouse cursor across a graphical screen supports the notion of the interface as a picture in 2D space. Navigation must be safe so that navigation is orthogonal to manipulating the user interface.

After users are able to navigate the auditory interface and identify the objects within it, they need the ability to **manipulate** those objects to accomplish their tasks. The most common manipulation is the ability to select an object whether it is a menu button or a text field. In the graphical interface, selection is generally accomplished by clicking on a mouse button. When we manipulate an interface, changes in the interface convey **feedback** as to the ramifications of our actions. The auditory interface must also provide conventions for manipulating the interface and providing feedback to the user.

Conveying Auditory Objects

To convey the contents of the auditory interface, it is necessary to convey the types of objects in the interface as well as attributes and affordances of those objects.

Conveying Object Identity

Numerous strategies for conveying objects in auditory interfaces have already been suggested by previous work. Possible strategies include using speech, pure tones, earcons, or auditory icons. For example, an auditory cue to convey a text-entry field could be:

- A synthesized voice saying "text-entry"
- A pure tone such as G-sharp (~ 415.3047 Hz)
- A musical timbre of a violin
- The sound of an manual typewriter

Each of these approaches has advantages and disadvantages. The speech message is unambiguous and reasonably efficient, but may be confused with other speech messages, i.e. reading the label on the field. A pure tone is easy to produce and takes minimal time to hear, but may be confused with other pure tones. Also the mapping of the note G-sharp to a text-entry field would be difficult to remember. Various musical timbres would also be easy to produce, and would be easier to discriminate than pure tones, but again, the mapping from violin to text-entry is hardly intuitive.

This design is based on the premise that auditory icons [14] offer the most promise for producing discriminable, intuitive mappings. In the previous example, the sound of an old-fashioned typewriter maps easily to a text-entry field. The user is reminded of typing or entering text. In general, the use of auditory icons mimics how information is conveyed in graphical interfaces. We recognize many objects in graphical interfaces by their physical appearance. Sometimes concrete representations are used such as the picture of a trashcan. Abstract icons also leverage our understanding of the physical world. Although Motif push buttons do not look like button controls in the physical world, they look pushable. Likewise, an auditory icon may not sound like a real push button, but the sound may indicate an object that can be pushed.

Two alternate design strategies that were considered and discarded were using speech or earcons. Synthesized speech is required for presenting textual information in the graphical interface. This information

is domain-dependent, such as the text in an electronic mail message or the labels on a pull down menu. By relegating speech to domain-dependent information, and respectively relegating nonspeech cues to domain-independent information, the user can more easily separate these classes of information[1]. The structured combinations of musical sounds employed in earcons[2][3][4] have been successfully used in providing access to mathematical equations for blind users[27]. That use of earcons was especially compelling since the natural prosody for reading mathematical equations mapped well to the rhythm of presenting successive earcons. In Mercator, the primary role of the auditory cues is to convey the types of objects in the graphical interface. We concluded that iconic, everyday sounds would be more intuitive than abstract, musical sounds.

In Mercator, we use a set of auditory icons to convey the identity of various interface objects. Some auditory icons are fairly concrete like the typewriter and the printer, while the sounds for various buttons are more abstract. The following table provides a listing of some of the auditory icons used in Mercator. The selection of sounds was based on a series of experiments exploring how people describe sounds and how they map concepts in graphical interfaces to sounds. These experiments are discussed in [23][24].

TABLE 1: Using Auditory Icons to Represent Interface Objects

Interface Object	Sound
Editable text area	Typewriter, multiple keystrokes
Read-only text area	Printer printing out a line
Push button	Keypress (ca-chunk)
Toggle button	Pull chain light switch
Radio button	Pa pop sound
Check box	One pop sound
Window	Tapping on glass (two taps)
Container	Opening a door
Pop-up dialog	Spring compressed then extended
Application	Musical sound

Conveying Object Characteristics

From our model of the graphical interface, we know there are many characteristics of the interface objects that we need to convey to the user. The use of auditory icons often serves to convey the affordances of the objects as well. For example, the typewriter sound should convey the affordance of entering text just as the push button sound helps convey the notion of pushing. But there are other attributes of objects we need to convey such as its label, whether it is greyed out, and its relative size.

Text-based attributes can be presented via synthesized speech. For example the auditory icon for a push button can be presented simultaneously with its text label. Other attributes can be presented by modifying the base auditory icon.

Auditory icons are not limited to simply reflecting categories of events and objects, but can be *parameterized* to reflect their relevant dimensions as well. For example, the auditory icon for a file can be manipulated to convey the size of the file [15]. Gaver's techniques for parameterizing auditory icons are similar to the *filtears* described by Ludwig, Pincever and Cohen [19][20]. We used the following filtears because they could process sounds in real-time[2]:

1. Since speech sounds are often less ambiguous than nonspeech, everyday sounds, speech output plays a role in supporting first-time users. We use redundant speech output to help users learn the meaning of the different nonspeech cues. The design of user levels is discussed later in this paper.

- Muffling

  High frequency energy in the auditory cue is removed, causing the cue to sound deeper in pitch with reduced intensity.

- Thinning

  Low frequency energy in the auditory cue is removed, causing the cue to sound higher in pitch with increased intensity.

By combining these filtears with modifying the overall intensity of the sound, we can create the impression of an auditory object being selected or greyed out.

Since muffling or thinning a sound affects our perceived pitch of the sound, we use these filtears to modify other auditory icons where the pitch of the auditory icon can be associated with its size or spatial location. If we strike two metal bins where one bin is much larger than the other, the sound of the larger bin will have a lower perceived pitch. Containers are objects in Mercator that group other objects, such as a collection of push buttons. The auditory icon for a container is an opening door. We modify this sound to indicate the number of items in the container. We use the same technique for text areas, so that the perceived pitch of the typewriter is based on the number of lines.

Sometimes it is helpful to convey the spatial location of an object or its position in a serial order. We modify the cursor sound to indicate how many lines down the cursor is in a textual list. We also slightly modify the sounds of grouped buttons indicating a button's location in the serial order. The modification is slight because extreme modifications are reserved for conveying the selection state of a button (selected, normal, greyed out).

TABLE 2: Manipulating Auditory Icons to Convey Object Attributes

Object	Attribute	Description of Filter
Button controls	Selection state (highlighted, normal, or greyed out)	Thinning and increase intensity for highlighting, muffling and decrease intensity for greyed-out
Text area	Number of lines	Lower pitch maps to greater number of lines, use muffling or thinning
Container	Number of children	Lower pitch maps to greater number of children, use muffling or thinning
Cursor	Location in serial order	Lower pitch maps to greater number in order, use muffling or thinning

Navigation

In addition to recognizing individual objects, the user must be able to navigate the space of the interface. The controls for navigation must support the user's mental model of the auditory interface. For example, moving the mouse cursor across a graphical screen supports the notion of the interface as a picture in 2D space. Since the conceptual model of the auditory interface is a hierarchical structure, the navigation controls should map to moving throughout that structure. For the controls to feel *automatic*, it is necessary for the meanings of the controls to be consistent throughout the interface, just as moving a mouse is consistent across the screen. This design is in contrast with typical phone interfaces where the meaning of the control (*Press 1 to do this*) is often context dependent.

Another comparison to mouse navigation is that navigation must be safe in the sense that it is orthogonal to manipulating the user interface. When users move a mouse across the screen, the interface may respond to

2. Facilities for muffling and thinning audio samples, as well as for playing, mixing and interrupting sounds was provided by NetAudio II, a tool developed by David Burgess [6].

give more information, but users are generally safe from triggering potentially harmful events such as stopping or starting an application. To support this separation, the navigation controls need to be distinct from the controls used to select or otherwise manipulate objects.

In Mercator, at the simplest level, the user uses the arrow keys on the numeric keypad to navigate a tree structure that corresponds to the condensed, hierarchical model of the graphical interface. The user presses up and down to move in and out of groups of objects and presses left and right to move within groups of objects. When the user moves to an object, they hear the auditory icon (possibly filtered) for that object. If the user attempts to move in a direction where no object exists, e.g. moving right when you are at the end of a cluster of push buttons, they hear a simple error sound of a ball bouncing against a wall. The premise is that the users reinforce their mental model of the auditory interface since the navigation is explicitly based on the hierarchical structure.

For the graphical interface pictured in Figure 3: "Annotated Graphical Interface" whose respective tree structure in shown below, a user navigating from the top of the structure to the push button "delete" would hear:

[1] (Down) Tapping on glass
[2] (Down) Opening door
[3] (Right) Printer "System Mail Box"
[4] (Right) Printer "Current message header"
[5] (Right) Typewriter
[6] (Right) Opening door
[7] (Down) Key press "next"
[8] (Right) Key press "delete"

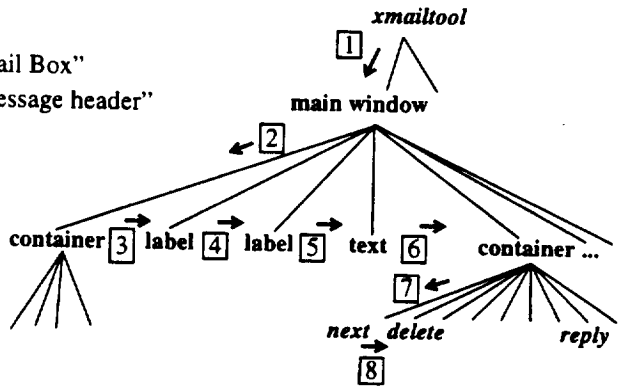

FIGURE 6: Navigation Based on Application Structure

This technique would be tedious if the user had to listen to the entire auditory icon each time they moved to an object. Although the auditory icons are short, average of one second, they are interruptible within approximately 50 ms. This set-up allows the user to quickly move throughout the interface. Also, it is important to remember that the navigation controls are consistent throughout the interface. Expert users seem to exhibit a form of muscle memory where they quickly press a sequence of keys to jump to parts of the interface. Some users even orient themselves by quickly moving to an "edge" in the tree structure, hearing the out-of-bounds sound, and then proceeding. Overall the feel of the navigation is quick and responsive.

Navigation Shortcuts

The persistent image of the graphical interface coupled with mouse input allows sighted users to quickly move from one portion of the interface to another. Even though Mercator users can quickly move throughout the interface, it is beneficial to provide keyboard shortcuts for expert users. One useful shortcut is the ability to move to the beginning, or end, of a group of objects. This jump is accomplished by hitting the 1 and 3 key, respectively, on the numeric keypad. The user hears quick snippets of the auditory icons for the objects that are "passed over" by using the short-cut.

Although the user could switch between applications by navigating to the "top" of an application and over to the next application, the user can press the right or left arrow key coupled with the Shift or Alt key to switch between applications. When the Alt key is used, the user is moved to the "top" of the next application. When

the Shift key is used, the user is moved to their last location in that application saved from the last time they used that application. This control helps the user recover their working context within an application. When the user switches between applications, they hear a paper flipping sound that should remind the user of switching between tasks, as well as the windows that are popping to the front of the screen contents. The new application name is announced with a message, such as, "Framemaker is the current application."

The user can also set hot keys for the row of keys above the numeric keypad. These keys can be used to move to a designated spot in an application interface that is specific to that application. The numeric keypad, annotated with the navigation controls, is pictured in the following figure.

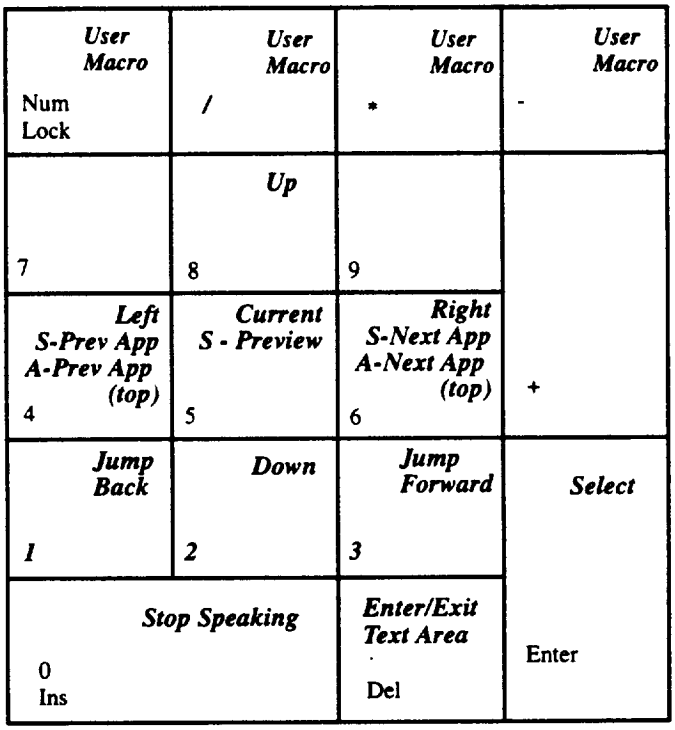

FIGURE 7: Mercator Navigation Controls

Although the navigation short-cuts assist the user in moving quickly throughout the hierarchy, they still do not afford the same freedom as quickly moving the mouse from one part of the screen to another. Different interaction styles not explored in this research include using a spatial model for the interface where the user could operate the mouse to move from one portion to another. Likewise, a tactile interface representing the tree structure could be used to provide a persistent overview as well as a medium for large jumps in the interface.

Auditory Preview

One limitation of auditory interfaces is the difficulty in presenting an overview of the interface contents. When sighted users look at a graphical interface, their eyes can quickly scan the interface to get a rough determination of its contents. Sometimes they can tell if they are where they want to be by the visual features of the interface. This technique applies to reading text as well. Robert Steven's [27] design of an auditory preview of mathematical equations can be applied to previewing portions of the Mercator interface. An auditory preview is simply short snippets of auditory icons that are played in quick succession. By the

overall length and diversity of sounds in the preview, the user gets a rough sense of the contents. The user can ask for previews of any group of objects, for example, objects grouped in a container or in a pop-up dialogue.

Sometimes the user does not need an auditory preview, but simply needs a reminder about the current object. In a visual interface, we can look away and then look back, regaining our visual focus. A user of an auditory interface may also need to regain the auditory focus. By pressing the 5 key, the user hears the auditory cue (may be a combination of nonspeech and speech output) for the current object. The inclusion of this feature is a simple example of learning from user feedback. The first Mercator interface did not include this control, and users (including us) would navigate away from and then back to the current object to regain the auditory context.

Manipulating the Interface and User Feedback

Up to this point, the description of the user interface has focused on the user perceiving and navigating the contents of the auditory interface. The next step is allowing the user to manipulate the interface. In this section, we describe how the users manipulate Mercator interfaces, as well as the feedback that the user receives from Mercator. The auditory feedback cues used in Mercator are summarized in Table 3: "Nonspeech Auditory Feedback in Mercator."

Selection

A principal action that users perform with graphical interfaces is selection. The action is typically accomplished by clicking (or double-clicking) on an object with the mouse. Since the Mercator user is working with the keyboard and not with the mouse, mapping selection to a keystroke reduces the distance that the users's hand must move. In Mercator, pressing the Enter key on the numeric keypad is mapped to selecting an object. Mercator can determine what mouse events the application expects (e.g. single or double click) and then simulate those events for the application.

Given the limitations of manipulating sampled sounds, creating pairs of sounds for {this is a push button, you just pushed a push button} was too difficult given the set of auditory icons used in Mercator. If the user successfully selects an object, the user will hear a short sound akin to someone ripping a batch of papers. This sound was chosen because it seemed to imply that something was happening, indicating to the user that the selection event had taken place. Since few users could actually identify the sound as ripping papers, they did not express any negative connotations about the sound. A longer discussion of the action-oriented content of sounds is presented in [23][24].

Pop-up Windows

Pop-up windows are an interesting case of the content, structure and focus of the interface changing almost instantaneously. When a pop-up window appears, the space of the interface (its content) is now augmented with the contents of the pop-up window. Likewise the structure of the interface, and our hierarchical model, is augmented by the structure of the pop-up. Often the input focus of the interface is moved to the pop-up as well, for example, modal pop-ups that require users to confirm or cancel an action before proceeding.

In graphical interfaces, pop-up windows capture the user's attention by being drawn on top of the other windows. In Mercator, whistling sounds are used to notify the appearance or disappearance of a pop-up window. A whistling sound with a rising pitch indicates that a pop-up has appeared, while a descending pitch indicates that a pop-up has disappeared. If the input focus is shifted to the pop-up, the user is moved to that location in the application tree structure. This move is indicated by the auditory icon for the pop-up window, a springy sound. There is a deliberate attempt to reinforce the terminology of pop-up window with these sounds. Both the whistling and spring sounds help form the illusion of something popping up in front of you.

When the user dismisses a pop-up, they are placed in their original location, where they were before the pop-up appeared. For example, in the Figure 3: "Annotated Graphical Interface", when the user presses the reply push button, they hear the following sounds as the pop-up appears on the screen:

"Rip" the selection is successful

"Whistle-up" the pop-up appears on the screen

"Spring" the user is moved to the top of the popup structure

If the user navigated to the cancel button, selecting that button and thereby dismissing the pop-up, they would hear:

"Rip" the selection is successful

"Whistle-down" the pop-up disappear from the screen

"Ca-chunk" "Reply" the user is moved back to the reply push button

Pop-up windows are stored in the interface model as descendants of the uppermost node of the application tree structure since they are perceived as separate windows on the screen. When the pop-ups are not modal, the user can navigate up out of the pop-up and back to the main application structure. If the pop-up is modal, such as requiring a confirm or cancel operation, the user is not allowed to navigate out of the pop-up, retaining the semantics of the interface.

TABLE 3: Nonspeech Auditory Feedback in Mercator

Action	Nonspeech Auditory Feedback
Selection	Ripping papers
Switching between applications	Paper shuffling
Navigation error	Ball rebounding against wall
Entering text mode	Rolling / rocking sound (drawer pulled out)
Moving edit cursor in text area	Click (pitch based on position in text)
Popup appearing / disappearing	Whistle up/down
Application connecting to Mercator	Winding
Application disconnecting	Flushing

Interacting with Text Objects

Screen readers for text-based interfaces, such as the command line interface to DOS, have existed for many years. These interfaces have formed a set of standard requirements for reading and manipulating text areas[1]. One requirement is support for two "cursors," an edit cursor that is located at the insert position in the text, and a review cursor that can be moved independently to read portions of the text. Operations for moving and synchronizing the cursors are coupled with operations for reading text by character, word, line, sentence and paragraph. Different filters are used to parse and pronounce the text based on the current task requirements. For example, a Unix filter can be used so that the command:

more dissertation.text | grep Mercator

would be read as:

more dissertation dot text pipe grep Mercator

1. Although there is not a paper detailing requirements for text-based screen readers, we were able to determine the needed functionality by examining existing screen readers and talking with blind computer users.

To review a text area, the user is required to enter "text mode" by pressing the ./Del key. For example, when the user navigates to a text area (hearing the typewriter sound), they then press the ./Del key to enter text mode. This operation is accompanied with a rolling/rocking sound to indicate moving into a different state. While in text mode, the keys on the numeric keypad are mapped to operations for reviewing text. The users can return to navigating the interface by exiting text mode, again pressing the ./Del key and hearing the rolling sound.

Some of the commands provided in Mercator for reading and manipulating text are summarized in the following figure.

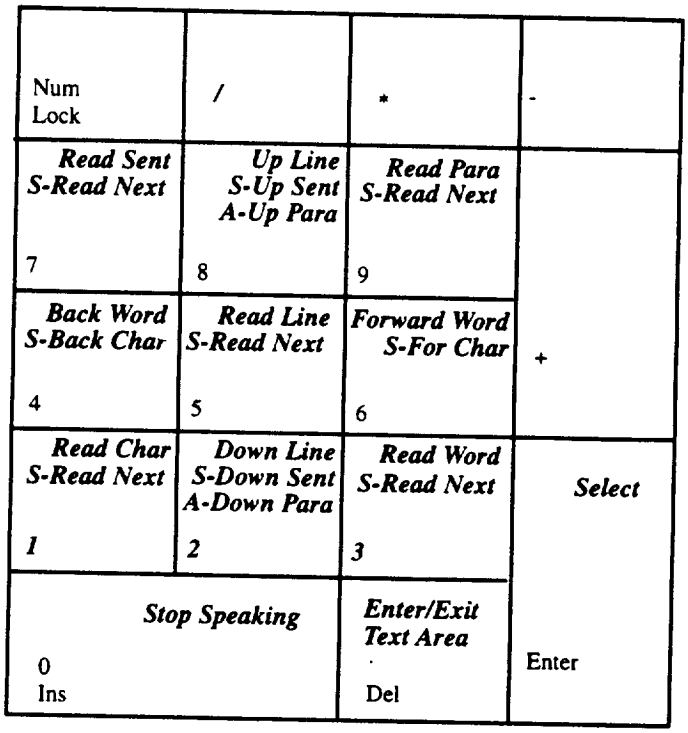

FIGURE 8: Mercator Text Controls

User Levels

Based on experience with demonstrating and evaluating Mercator, it became clear that the interface could be modified to support the transition from a novice to expert user. The tcl interface code was easily extended to support three user levels (Novice, Intermediate, Advanced). The primary modifications focused on information presented to the user when they navigated to an object, and when they requested information about an object. Based on observations of people using Mercator, three stages of learning became apparent.

- **Recognizing auditory icons**

  The users learned the sounds for push buttons, text areas and so on.

- **Parameterized auditory icons**

  The users learned how the auditory icons are manipulated to convey attributes of objects such as a push button being greyed out.

- **Understanding modes**

  Users learned that they have to enter "text mode" to review the contents of a text area.

The current user level determined the amount of redundant speech information. What the user would hear, per user level, after navigating to a greyed out push button labeled undelete, is shown in the following table.

TABLE 4: Interface Output for Push Buttons Per User Level

Novice	*Push button (muffled ca-chunk sound) undelete greyed out*
Intermediate	*(muffled ca-chunk sound) undelete greyed out*
Advanced	*(muffled ca-chunk sound) undelete*

When the user asks for information about an object by pressing the 5 key, they hear the information corresponding to one level less experienced than their current setting. This strategy helps users transition between levels. For example, a user can switch to operating as an Intermediate, but still get additional information for objects that they have forgotten or have not encountered.

ASSESSING MERCATOR'S INTERFACE

During the course of this research we have utilized many methods for assessing Mercator's design including discussing our design with users and other designers, observing people using Mercator as well as observing how people teach others to use Mercator, and measuring the performance of people conducting specific tasks. To collect quantitative data on the learnability of Mercator, we measured how quickly sighted users reached peak performance in a specific task of reading and replying to email messages using a graphical email application. One motivation for using sighted people in this experiment is that we also examined the effects of transitioning between using the graphical and auditory versions of the same application.

We have also compiled reactions by blind users that we have received over the past three years. We did not perform controlled experiments with blind users for two reasons. First, previous experience with computers appears to be an overriding factor in how well blind users perform with screen readers for graphical interfaces. It would have been difficult to control previous experience so that performance data would be meaningful across subjects. Second, the available sample of blind users in the Atlanta area generally had no computer experience. In contrast, users attending conferences for assistive technology, generally had comparable experience with computers and were motivated to use graphical interfaces. We discussed Mercator's design with potential users at over ten conferences that included an emphasis on assistive technologies. At three of the these conferences, Mercator was available for use over multiple days among the product exhibits. From these experiences, we have summarized favorable and critical assessments of Mercator interfaces.

Measuring Performance with Mercator

Having already observed that blind users could learn to use Mercator, we wanted to assess how well sighted users performed with Mercator for two reasons. First, we needed a controlled setting in which we could measure the time needed to learn to use Mercator. Based on demonstrations with blind and sighted users, it appeared that computer literate sighted users took longer to learn the interface than computer literate blind users, but that the stages of learning were the same.

Second, we wanted users to contrast their use of a graphical interface and the Mercator-derived auditory interface. One hypothesis was that experience with the graphical interface would be beneficial in using the auditory interface since the two interfaces share the same structure.

In order to assess users' performance with the auditory interface, as well as determine the effects of prior experience with the graphical interface, test subjects worked with graphical and auditory versions of the application xmailtool. A screen shot of the graphical interface is shown in Figure 3: "Annotated Graphical Interface".

Quantitative data was calculated from analyzing activity logs. The logs indicated each time an event had occurred in the interface, such as moving to a new object, entering or exiting text mode, or selecting an

object. With the subjects' consent, we videotaped the sessions including training and debriefing in addition to the test trials.

Seventeen subjects worked with combinations of the graphical and auditory interfaces. The subjects were randomly divided into four groups as shown in the following table. The group designation determined which interfaces they used, and in what order. For example, in Group 2, the subjects started with the graphical interface, but switched to the auditory interface after four trials. The subjects in Group 3 only used the auditory interface.

TABLE 8-1 Experimental Design

	Part A (4 trials)	**Part B (4 trials)**
Group 1 (3 subjects)	Graphical	Graphical
Group 2 (6 subjects)	Graphical	Auditory
Group 3 (5 subjects)	Auditory	Auditory
Group 4 (3 subjects)	Auditory	Graphical

In each trial, the subject selected, read and replied to three specified email messages.

The training for the experiment was conducted in three stages.

- **Description of the Task**

  I told the subjects the details of the task they were to perform, namely that they were to locate, read and reply to three specified email messages. I explained that in each message was a test phrase that they would need to type into their reply.

- **Description of the Interface**

  At this point, I either described the graphical interface or the auditory interface. I explained how to navigate the interface and how to select objects.

  I also showed the subjects a diagram of the common structure of the graphical and auditory interfaces similar to the diagram in Figure 5: "Partial Tree Structure Representation of Graphical Interface".

- **Demonstration of the Task**

  I demonstrated replying to one email message. I went through all of the steps including writing and sending the reply.

Learning the Auditory Interface

The most promising result of the experiment is that all of the subjects were able to learn how to use the auditory interface. The average time to complete a trial sharply decreased after one trial with peak

performance achieved around the fourth trial. Another important result is that the variance in time taken sharply decreased after one trial (average of 401.33 to average of 88.96).

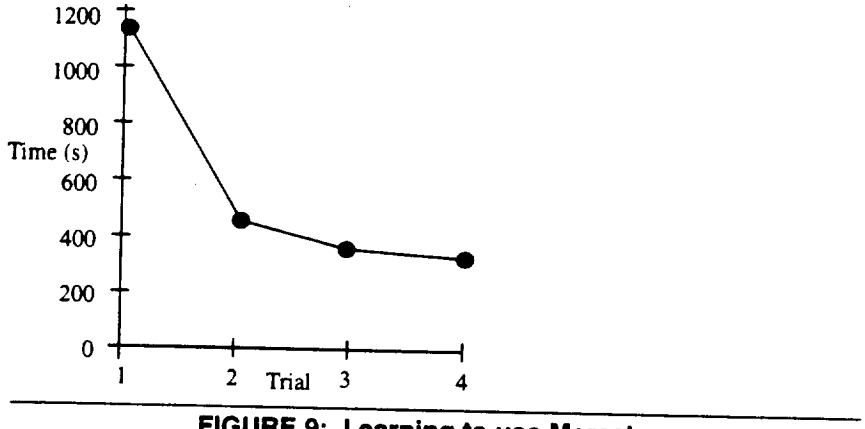

FIGURE 9: Learning to use Mercator

Stages of Learning

So what did the subjects learn? From observation and inspection of the data, it appears that four concepts were acquired in the approximate order:

1. Basic Auditory Icons

Since the subjects had only heard a brief demonstration, they needed to spend some time recognizing and learning the auditory icons. Although ten sounds were needed in this interface, the subjects never asked what a sound meant. They seemed to use the 5-Info key to hear the auditory icon coupled with redundant speech information until they learned the meaning of the auditory icon.

2. Navigation

The biggest hurdle in using the interface is understanding the hierarchical navigation scheme. Subjects who had never seen the graphical interface had to learn that the down and up keys took them in and out of groups of objects. Improved navigation times greatly contribute to overall improved performance times. In part, improved navigation times seemed to be impacted by how safe the users felt. As users realized that they could navigate the interface without causing unwelcome consequences, they increased their rate of input, "bouncing off the walls" when they went too far in any direction.

3. Parameterized Auditory Icons

As the subjects continued using the interface, it became apparent that they were learning to listen for the pitch differences between auditory icons of the same class. For example, the reply push button is in the container with 18 children. This container has a deeper sound than other containers in the interface. Likewise the text areas with the headers and message are much larger than the text area with diagnostic output. Subjects learned to listen for the container and text areas with a lower pitch helping them locate these objects faster and more reliably.

4. Text Mode

A common guideline in human-computer interaction is to avoid modes in the interface. Mercator has one mode and it proved problematic. When users navigate to a text area, they need to enter text-mode so that the numeric keypad can then be used to navigate the text as opposed to navigating the rest of the interface. Often a subject would reach a text area, but not remember to switch into text mode. Common guesses were selecting the text area and trying to navigate down into the text area (not a bad idea!).

Transitioning between the Graphical and Auditory Interfaces

One way to demonstrate that the auditory interface captures critical characteristics of the graphical interface is to look for a *transfer effect* when the user transitions from using the interface in one modality to using the interface in the other modality. For example, if the user has experience with the graphical interface, this experience should help the user learn the auditory interface. Unfortunately quantitative measurements did not demonstrate that such an effect took place. There are two reasons related to the experimental design that may help explain why the transfer effect was not evidenced:

- **Exposure to Interface Structure**

  During the training, I showed all of the subjects a diagram of the interface structure. Users of the graphical interface paid little attention to the diagram. In contrast, users of the auditory interface studied the diagram and indicated that they would have preferred to consult the diagram during the task. The information in that diagram is in essence the information that should cause a transfer effect. Experience with the graphical interface should give the user information about the structure of the interface. By showing the diagram to the users of the auditory interface, I accidently gave those users the same information that the transfer effect is based on. Therefore the effect was hidden by the improved performance of the users of the auditory interface.

- **Graphical Task Too Easy**

  Performing the task with the graphical interface required little cognitive effort. The performance times increase over the trials is likely due to boredom. The subjects spent most of their time trying to determine what I was actually testing them on. One subject asked me if I was manipulating the lights in the room. Since they did not have to think about the task, they internalized little information about the content of the graphical interface.

During the debriefing, subjects who first used the graphical interface and then used the auditory interface made three interesting observations:

- **Exposure to Graphical Interface Helped in Using Auditory Interface**

  Although not evident in the quantitative analysis, the subjects reported that their experience with the graphical interface was helpful in understanding the auditory interface. Aspects of the graphical interface that were helpful included knowing the objects in the interface, the spatial ordering of the objects, and the relative sizes of objects.

- **Subjects Needed to Update their Simple Model of the Interface**

  Although subjects reported that their exposure to the graphical interface was helpful, they remarked that they needed to form a more complex model of the interface when working with the auditory interface. They did not describe forming a new model, but augmenting their simple model with more information. For example, in the graphical interface, the subjects could easily ignore a number of objects in the interface. Since they had to navigate past these objects in the auditory interface, they needed to augment the interface model with these objects.

- **Initial Transition Between Spatial and Hierarchical Was Difficult**

  Since the subjects had a fresh visual image of the graphical interface in their minds when they began working with the auditory interface, they typically tried to navigate the interface based on the spatial layout. Most of the interface objects are arranged top to bottom in the graphical interface, but since they are sibling objects, they are accessed by moving left and right in the auditory interface. Navigation errors from trying to move in spatial directions generally disappeared during the first trial.

Observations by Blind Users

Reactions to Nonspeech Auditory Cues

Blind users have expressed an overwhelming positive response to the use of everyday sounds in screen reader interfaces. As noted previously, users of current screen readers have difficulty separating interface information, such as "Push Button'" from application information, such as "Edit," when both types of information are presented with speech or braille. When blind users were able to work with and listen to the Mercator interface, they remained impressed with the use of everyday sounds. In addition to particularly liking the typewriter and whistle sounds, in contrast to sighted users, blind users liked the container sound and were rarely confused about its use. Users commented that the filtering of the auditory icons was subtle, noting that many designers unnecessarily exaggerate changes in audio.

Reactions to Hierarchical Interface Structure

In contrast to the use of everyday sounds, blind users were skeptical about hierarchical navigation schemes as presented during design briefings. The general consensus was that they needed to "know what was on the screen" since that was what their sighted counterparts used. Only after using Mercator, did users express their preference for this scheme.

Users have requested that Mercator allow them to print out information about the structure (object hierarchy) of an interface using a braille printer. Users experimenting with this strategy refer to consulting the constant tactile image while exploring the auditory interface. The tactile image seems to provide some of the functionality that the constant visual image provides to sighted user.

A new screen reader also uses an underlying hierarchical model. The system, called Virgo, transforms Microsoft Windows interfaces into braille interfaces. Instead of using auditory icons, the first two braille cells contain a code that represents the type of objects, and the remaining braille cells contain the label and highlighting information.

Meeting Goals for Screen Reader Design

After discussing the benefits that graphical interfaces provide for sighted users, we outlined six goals for transforming graphical interfaces into auditory interfaces. Given the auditory interface design discussed in this paper, how well does Mercator meet those goals?

- **Access to functionality**

  By providing general strategies for representing graphical interfaces with auditory interaction techniques, Mercator provides transparent access to applications for word processing, electronic mail, calendars and so on. In these text-based interfaces, spatial information in the interface is generally mapped to structural information. One exception is when domain specific information such as the text in a document is searched spatially using the controls for manipulating and reading text.

  One advantage of Mercator is that all objects in the interface are treated as first class objects. In contrast to current screen readers, users do not have to define special view areas to access portions of the interface that do not accept user input such as message bars.

- **Iconic representations of interface objects**

  When possible, interface objects are grouped into the same discrete objects that sighted users perceive. The identify and attributes of these objects are conveyed with auditory icons. Like their graphical counterparts, auditory icons leverage knowledge of the real world in presenting interface output.

- **Structural organization**

  A central motivation in Mercator's design is to convey the underlying structure in graphical interfaces. The groupings of objects, conveyed with visual cues in the graphical interface, are made evident as the user navigates into, within, and out of object groups. These groups help clarify the functionality of individual objects.

- **Direct manipulation**

  As in graphical interfaces, users directly interact with objects in the interface and interface output is conveyed via the objects. In as much as the graphical interface provides objects that match how user's conceptualize tasks with the application, Mercator provides a direct manipulation interface.

- **Spatial arrangement**

  A primary difference between Mercator and commercial screen readers is that Mercator is based on a hierarchical model of the graphical interface as opposed to a spatial model. Mercator, however, does provide information about the spatial attributes and layout of the graphical interface. The relative sizes of objects are conveyed by manipulating their base auditory icons. Information about the layout of text is conveyed by modifying the sound of the edit cursor as it is moved throughout the text. The layout of objects helps determine their ordering in the auditory interface.

  Nevertheless, information about the layout of the interface is lost in this representation. Likewise, users are not able to arrange application windows along spatial dimensions. This design trade-off was made to offset existing usability problems with commercial screen readers.

- **Persistent presentation**

  A benefit of visual interfaces is that they exist in physical space and can be reviewed over time creating a surrogate short-term memory for recalling the contents of the user interfaces. This type of persistent presentation is difficult to achieve in a complex auditory interface where multiple continuous sounds are confusing and distracting. To improve the user's scanning capabilities, we provide the preview facility. The short snippets of the auditory cues help convey portions of the interface, and is sufficiently succinct to confirm the user's location in the interface.

  Some blind users have experimented with using braille printouts of the interface structure. The users refer to this constant tactile image while exploring the auditory interface.

FUTURE WORK

This design is effective for blind users working with text-oriented applications such as word processors, electronic mail and other menu and form-based interfaces. The challenge of providing access to more graphical applications such as drawing programs remains. One area of future research is incorporating the use of a tactile display. The auditory and tactile displays could be used to create complementary presentation of the graphical interfaces. The tactile display would help offset some of the limitations of the auditory interface by providing a constant presentation of the interface as well as supporting large moves across the space of the interface.

In many ways, this research addressed an important, but overly constraining problem of transparent access to graphical applications. Once the constraint of transparency is relaxed, one could imagine combining aspects of spatial and conversational interfaces into the default hierarchical interface to leverage domain-dependent interaction. The potential of adding voice interaction is especially compelling. We have extended the Mercator architecture to include voice as a potential input source, but we have not furthered explored its use. The inclusion of a spatial model could aid in providing access to a broader range of applications that inherently include spatial content such as drawing and map-based tasks.

Sighted computer users could also benefit from auditory representations of graphical interfaces while performing eyes-busy tasks such as driving, performing maintenance on a airplane, or inspecting a manufacturing plant. The needs of these users will be different however. For instance, supporting mobility will likely be a key requirements. In these cases, improving the flexibility of conversational interfaces may provide the most promise.

ACKNOWLEDGEMENTS

The Mercator project represents a multi-year, multi-person effort. Thanks go to Keith Edwards, Tom Rodriguez, Kathryn Stockton, Ian Smith, Sue Liebeskind, Sue Long, Kevin Chen, Will Luo and Stacy Ann Johnson for their design and implementation contributions.

This research was supported by Sun Microsystems, the NASA Marshall Space Flight Center, the National Security Agency, and Georgia Tech.

REFERENCES

[1] outSPOKEN, The Talking Macintosh Interface. User Manual. Berkeley Systems. 1989.

[2] Blattner, M., Glinert, E. P., and Papp, A. L., III. Sonic Enhancements for 2-D Graphic Displays, *Auditory Display: Sonification, Audification and Auditory Interfaces*, edited by G. Kramer, SFI Studies in the Sciences of Complexity Proc. Vol. XVIII, Addison-Wesley, 1994, pp. 447-470.

[3] Blattner, M. M. and Greenberg, R. M. Communicating and Learning Through Non-Speech Audio. In *Multimedia Interface Design in Education*, edited by A. Edwards and S. Holland, Springer-Verlag, NATO ASI Series F, 1992, pp 133-143.

[4] Blattner, M. M., Sumikawa, D. A, and Greenberg, R. M. Earcons and Icons: Their Structure and Common Design Principles. Human-Computer Interaction 4(1), 1991, pp. 11-44.

[5] Boyd, L.H., Boyd, W.L. and Vanderheiden, G.C. The graphical user interface: Crisis, danger and opportunity. In *Journal of Visual Impairment and Blindness*, December 1990, pp. 496–502

[6] Burgess, D. The NA3 Audio Server. Final report to Sun Microsystems. 1993.

[7] Day, G. Personal communication, 1995.

[8] Edwards, A. D. N. Graphical User Interfaces and Blind People, Proceedings 3rd International Conference on Computers for Handicapped Persons, Vienna, July 1992, pp 114-119.

[9] Edwards, A. D. N. Evaluation of Outspoken software for blind users, University of York, Department of Computer Science Technical Report YCS150, 1991.

[10] Edwards, A. D. N. Modeling blind users' interactions with an auditory computer interface. *International Journal of Man-Machine Studies*, 1989, pp. 575-589.

[11] Edwards, W. K. and Rodriguez, T. Runtime Translation of X Interfaces to Support Visually-Impaired Users. In *Proceedings of the 7th Annual X Technical Conference*, Boston, MA, 1993.

[12] Edwards, W.K. and Mynatt, E.D. An Architecture for Transforming Graphical Interfaces. In the *Proceedings of UIST'94: User Interface Software and Technology Symposium*, Marina Del Ray, CA., Nov. 2-4, 1994, 39-47.

[13] Edwards, W.K., Liebeskind, S. H., Mynatt, E.D and Walker, W.D. A Remote Access Protocol for the X Window System. In the *Proceedings of the 9th Annual X Technical Conference*, Boston, MA, 1995.

[14] Gaver, W. W. Everyday listening and auditory icons. Doctoral Dissertation, University of California, San Diego. 1988.

[15] Gaver, W.W. Using and Creating Auditory Icons. In *Auditory Display: Sonification, Audification and Auditory Interfaces*, edited by G. Kramer, SFI Studies in the Sciences of Complexity Proc. Vol. XVIII, Addison-Wesley, 1994, pp. 417-446.

[16] Glinert, E.P. and York, B.W. Computers and People with Disabilities, in *Communication of the ACM*, 35(5) 1992, pp. 32-35.

[17] HumanWare, Artic Technologies, ADHOC, and The Reader Project. Making good decisions on technology: Access solutions for blindness and low vision. In *Closing the Gap Conference*, October 1990. Industry Experts Panel Discussion.

[18] Hutchins, E.L., Hollan, J.D. and Norman, D. A., Direct Manipulation Interfaces. In *User Centered System Design*, edited by Norman, D.A. and Draper, S.W., Lawrence Erlbaum Associates, Inc., 1986, pp. 87-124.

[19] Ludwig, L. L., Pincever, N. and Cohen, M. Extending the notion of a window system to audio. *Computer*, August 1990, pp 66-72.

[20] Ludwig, L. L. and Cohen, M. Multidimensional audio window management. *International Journal of Man-Machine Studies*, 34(3), 1991, pp 319-336.

[21] Ly, E. Chatter: A Conversational Telephone Agent. MIT Master's Thesis, Program in Media Arts and Sciences, 1993.

[22] Mynatt, E. and Weber, G. Nonvisual Presentation of Graphical User Interfaces: Contrasting Two Approaches," in the *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 1994.

[23] Mynatt, E. Designing Auditory Icons, In *Proceedings of the Second International Conference of Auditory Display, ICAD '94*, Sante Fe, New Mexico, 1995, pp. 109-120.

[24] Mynatt, E. Transforming Graphical Interfaces into Auditory Interfaces. Doctoral Dissertation, Georgia Institute of Technology, Atlanta. 1995.

[25] Norman, D. A. *The Psychology of Everyday Things*. New York: Basic Books. 1988.

[26] Schmandt, C. Phoneshell: the Telephone in Computer Terminal. *Proceedings of ACM Multimedia Conference*, August 1993.

[27] Stevens, R., Brewster, S., Wright, P. C., and Edwards, A. D. N. Design and Evaluation of an Auditory Glance at Algebra for Blind Readers. In *Proceedings of the Second International Conference of Auditory Display, ICAD '94*, Sante Fe, New Mexico, 1995, pp. 21-30.

[28] Stifelman, L. J., Arons, B., Schmandt, C. and Hulteen, E. A. VoiceNotes: A Speech Interface for a Hand-Held Voice Notetaker. In *Proceedings of INTERCHI '93*, ACM SIGCHI, 1993, pp 179-186.

[29] Yankelovich, N. SpeechActs & The Design of Speech Interfaces, in the *Adjunct Proceedings of the 1994 ACM Conference on Human Factors and Computing Systems*, Boston, MA, 1994.

[30] Yankelovich, N., Levow, G., and Marx, M. Designing Speech Acts: Issues in Speech Interfaces, Proceedings of CHI '95, Denver CO, May 8-11, 1995.

ULTRASONIX USER MANUAL

1.0 INTRODUCTION

1.1 About UltraSonix

Welcome to UltraSonix, a screen-reader, for X Windows applications. UltraSonix is a software system that allows you to use many X Windows applications even if you cannot see the screen. UltraSonix provides speech, auditory sound effects and braille output to represent the visual, graphical interface. UltraSonix also provides keyboard alternatives to the mouse pointing device.

1.2 Learning X Windows Applications

For many users, using UltraSonix may be their first experience with accessing graphical applications. Part of the learning process will be learning about how typical X Windows applications work. Graphical interfaces are unlike command-line interfaces such as DOS. This manual introduces concepts underlying graphical X Windows applications. Nevertheless, the best way to learn is through experimentation.

1.3 About this Manual

This manual is organized into 7 sections:

1.0 INTRODUCTION
  What you are reading now.
2.0 INSTALLATION
  Instructions for installing UltraSonix
3.0 INTRODUCING X WINDOWS
  A short review of the concepts underlying X Windows applications and GUI interfaces.
4.0 GETTING STARTED
  This contains an introduction into using UltraSonix and should require sufficient instructions for many beginning users.
5.0 BRAILLE OUTPUT
  Instructions for adding braille output to UltraSonix
6.0 CONFIGURATION
  Information for configuring UltraSonix
7.0 REFERENCE
  A summary of the commands used by UltraSonix as well as the sound effects.

2.0 INSTALLATION

This section describes how to install UltraSonix on your platform, and what the requirements of the target system are.

2.1 Platform Requirements

The UltraSonix screenreader software is designed to be run on Sun SPARCstation or SPARCstation-compatible computers running the Solaris 2.4 (or later) operating environment.

Hardware and operating system requirements are:

- At least 32MB of memory recommended for best performance.
- SPARCstation 2 or better recommended for best performance (a faster machine may be necessary if you plan on using software-based speech synthesis).
- 10MB free disk space.
- Solaris 2.4 or later.
- Either 8-bit or 16-bit audio hardware (standard on most Sun computers).
- A supported speech synthesis system (currently DECtalk DTC01, DECtalk Express, and Entropic TrueTalk are supported).
- Either version R5 or R6 of the X Window System. The OpenWindows 3.4 (or later) window system from Sun is recommended, and comes as the default on Solaris 2.4. If you do not run OpenWindows 3.4, your server must support the XTEST extension. We highly recommend using the OpenWindows server, however.

Optional supported hardware includes:

- A supported Braille terminal system (currently the Alva 3-20 or Alva 3-80).
- An external keyboard device (currently the Genovations device is the only supported external keypad).

To operate with the UltraSonix software, applications must have the following properties:

- Be written to the Xt Intrinsics toolkit, version R5.
- Be based on the Motif or Athena widget sets.
- Be dynamically linked against the X libraries.
- Not have the setuid bit set (Solaris security does not support the use of shared libraries from non-standard locations by setuid applications).

In order to compile the UltraSonix softare, you need the following:

- An ANSI-compliant C compiler (we recommend SPARCcompiler C version 3.0 or later).
- A C++ compiler with support for templates and exceptions (we recommend SPARCcompiler C++ version 3.0 or later).
- The Rogue Wave Tools.h++ library for C++ (this library comes with SPARCcompiler C++ version 3.0 and later).
- ANSI/POSIX-compliant header files and system call interfaces.

2.2 Installing the Software

The UltraSonix software is distributed in the "package" format. Packages are a standard mechanism for software distribution used in Unix SVR4. They provide features such as automatic versioning, dependency analysis, and consistent installation procedures.

Solaris provides two separate tools for installing software packages. The first is a graphical tool called swmtool. The second is the command-line pkgadd. Both are in /usr/sbin. Instructions for using each are provided below.

Before you begin installation, you must decide on a location for the software. We recommend installing in /opt (the standard location for "optional" software in SVR4). You can install UltraSonix anywhere however.

Note that the procedures below assume that you are running Solaris Volume Management (the Volume Manager controls the automatic mounting of floppies).

2.2.1 Installation Using SWMTOOL

(NOTE: Installation via swmtool is not currently supported.)

2.2.2 Installation Using PKGADD from a Tar File

Uncompress the file GTsonicx.tar.Z via the command

```
uncompress GTsonicx.tar.Z
```

and then untar it with the command

```
tar xvf GTsonicx.tar
```

This can be done in any directory which has more than approx. 8 meg. of storage. After this is done there should be a directory called GTsonicx.

Next, you must use the pkgadd command to install the UltraSonix software onto your system. pkgadd is located in the /usr/sbin directory and must be run as root (so that it can update various system databases which record information about what packages are installed on the system).

By default the package will be installed under the /opt directory. If you wish to use a different location, you must specify the location on the command line. To install in /opt, type the following command:

```
pkgadd -d <GTsonicx package location> GTsonicx
```

While we recommend installing the package in the default location, it is possible to install it in a different directory. The following command will cause pkgadd to prompt you for the location to install the package.

```
pkgadd -a none -á <GTsonicx package location> GTsonicx
```

The package will be copied to the directory specified by the -R option.

2.2.3 Installation Using PKGADD from Floppies

Insert the floppy labeled UltraSonix 1/5 into the floppy diskette drive. The exact command sequence you will follow depends on whether or not you're running the Solaris volume manager. (The volume manager is the software system on Solaris which handles mounting and unmounting of removable media. By default, the volume manager is always running.)

This document describes installation on systems running the volume manager.

To tell the volume manager to detect the presence of a floppy disk in the drive, type the following command:

```
volcheck
```

The installation procedure is similar to the one described above but the location of the package is now different:

```
pkgadd -d /vol/dev/aliases/floppy0 GTsonicx
```

While we recommend installing the package in the default location, it is possible to install it in a different directory. The following command will cause pkgadd to prompt you for the location to install the package.

```
pkgadd -a none -d /vol/dev/aliases/floppy0 GTsonicx
```

The package will be copied to the directory specified by the -R option.

As pkgadd runs it will prompt you to install the other four floppies as needed.

2.3 Directory Layout

In the following section, we assume that the package has been installed in /opt, the default location. If you have installed the package in a non-default location, simply prepend the directory you specified to pkgadd to "/opt" in the information that follows.

Once the package has been installed in /opt/GTsonicx, several subdirectories will be created. The layout of the directory structure is as follows:

```
/opt/GTsonicx/

    bin/
            alvad
            console
            dectalkd
            mercator-configuration
            mercator
            netaudiod
            runsonicx

    lib/

            R5/
                libX11.so
                libXt.so

            R6/
                libICE.so.6.0
                libX11.so.6.0
                libXext.so.6.0
                libXt.so.6.0
                libSM.so.6.0
                libXaw.so.6.0
                libXmu.so.6.0
```

R6.orig/
    libX11.so.6.0
    libXt.so.6.0

RAP/
    libRAPagnt.so
    libRAPclnt.so

loadables/
    Alva.so
    DectalkX.so
    NetAudio.so
    Dectalk.so
    Genovations.so
    Truetalk.so

scripts/
    <all tcl files>

sounds/
    <all sound files>

templates/
    Athena/
    Motif/
    Apps/
        <all templates, in subdirs>

etc/
    mercator.attrib
    mercator.config
    default.flt

apps/
    xmailtool-r5
    xmailtool-r6
    chitrivia
    pizza-tool

doc/
    ProcessMgmt
    briefing
    dectalk.txt
    truetalk.txt
    ScreenReader.txt
    config_files
    dectalkX.txt
    user-manual
    UsingFDInterest
    console.txt
    genovations.keypad
    WritingLoadables
    copyright
    install.doc

src/
    console/
        console.cc
    mercator-configuration/
        (all sources for mercator-config.)

Other included files which do not show up in the GTsonicx directory are:

- pkginfo
- prototype
- postinstall
- preremove
- copyright

In order to use UltraSonix, you must configure the system to tell it what hardware devices are attached to your workstation, and where various required system files are located. See Section 6.0, "Configuration," for more details on configuring the software.

After you have configured the system, UltraSonix should be ready to run.

3.0 INTRODUCING X WINDOWS

[Need general description of X Windows and concepts in GUI interfaces]
Need to define:
    Window
    Mouse
    Focus
    ...

4.0 GETTING STARTED

The best way to learn how to use UltraSonix is to start experimenting with a few applications. This chapters leads you through a simple tutorial for learning the basic operations of UltraSonix. It does not present every feature but tries to convey the feel of using this screen reader. Pointers to additional information are included as well.

4.1 UltraSonix Keypad

The majority of commands to UltraSonix are issued through the numeric keypad located at the far right of your keyboard. Here is the layout of the standard numeric keypad:

Num Lock	/	*	-
7/Home	8/Up	9/PgUp	+
4/Left	5	6/Right	
1/End	2/Down	3/PgDown	Enter
0/Insert		./Delete	

The + and Enter key are twice as long as the standard keys, while the 0/Insert key is twice as wide.

UltraSonix uses two major modes of operation. The first mode is navigation. In this mode, you can navigate to different objects in the graphical interface such as push buttons, windows and other applications. The second mode is for reading and editing text. In this mode, you can search through sections of text as well as enter textual input from the keyboard. In navigation mode, the keypad controls are as follows:

Unassigned	Unassigned	Unassigned	Unassigned
Unassigned	Up	Unassigned	Drag/Release
Left	Read Current	Right	
First	Down	Last	Select
Stop Audio		Text Mode	

These controls are explained further in the section titled Navigation.

In text mode, the keypad controls are as follows:

Unassigned	Cursor Modes Filter Modes	Disable
Read Sentence Up Line	Read Paragraph	Enable
Left Character Read Line	Right Character	
Read Character	Down Line	
Read Word	Select	
Stop Audio	Text Mode	

The text mode key is used to switch UltraSonix in and out of text mode. The text reading functions are explained further in the section titled Accessing Text.

Additional keyboard input is used, such as using modifier keys (shift, meta, alt and control) with the numeric keypad. These keyboard combinations will be explained in the appropriate section.

4.2 UltraSonix Console

When you start UltraSonix, a UltraSonix console is started for you. UltraSonix immediately places your focus in this application. The console allows you to control various parameters and settings in the UltraSonix software. Via the console, you can change speech rate, voice, user expertise level, and several other settings. There is also a text area where you can type commands directly to the UltraSonix process (this is primarily useful for debugging and testing of the software).

4.3 Running Applications with UltraSonix

To operate with the UltraSonix software, applications must be run in a special way. UltraSonix works by substituting its own special versions of the X Window System libraries in applications as they run. Applications that aren't run using these special libraries will not be "visible" to the UltraSonix software.

To run an application under UltraSonix, use the command "sxrun." Sxrun is a shell script that sets several environment variables and then executes the rest of the arguments on its command line as a command. For example, to run the "xmailtool" application, you would type the following command:

```
sxrun /opt/GTsonicx/bin/xmailtool
```

Type this now to start the xmailtool application; we will use this application as a demonstration in this tutorial.

4.4 Starting, Stopping and Switching Between Applications

As you start, stop and switch between applications, UltraSonix maintains a list of all your applications including the UltraSonix console as your first application. Your focus is always on one of these applications. When you start an application that UltraSonix knows how to provide access to, you will hear a winding sound (like someone winding a music box), then you will hear a music sound as you are moved to the new application. When you exit an application, you will hear a flushing sound, and then a music sound as you are moved to the previous application in the application list.

You can press alt-right and alt-left to switch between applications by pressing the alt key, holding that key down, while pressing the left or right arrow keys on the numeric keypad. When you switch applications, you will hear a paper shuffling sound, and then the music sound. Each time you hear the music sound, the speech synthesizer will announce the title of the application that is the new focus.

Summary of controls presented in this section:

Alt-right:	Move to the next application
Alt-left:	Move to the previous application

Summary of sounds presented in this section:

Winding:	Application starting
Flushing:	Application ending
Paper shuffle:	Switching between applications
Music:	Focus moved to an application

4.5 Navigation

At this point, you should have started UltraSonix and then started the application xmailtool from the UltraSonix console. After xmailtool has finished hooking up to UltraSonix, you will hear the music sound as your focus is moved to this application. This section will lead you through the basic steps of navigating an application interface by describing how to navigate the application xmailtool.

The basic premise behind UltraSonix's navigation controls is that UltraSonix maps the GUI interface into a tree structure. To explore the interface, you can walk up and down the interface's tree structure using a simple set of commands. The first command to learn is:

5:	Info about current location

By pressing the 5 key on the numeric keypad, you can hear information about your current location in the application interface. By pressing 5 now, you should hear the music sound and "xmailtool." (Quotes indicate synthesized speech.) The music sound indicates that you are at the top of an application tree structure.

The next commands to learn are:

8/Up arrow:	Go up one level to the parent of the current object
2/Down arrow:	Go down one level to the first child of the current object

Press 2 to go down one level. You should hear the sound of tapping on glass for a window (tink tink). If you press 5, you hear tink tink "Window"

Press 2 to go down one level. You should hear a container sound (opening door). Press 5 to hear opening door and "Container."

The next commands to learn are:

6/Right arrow: Go to the next sibling in a group of objects
4/Left arrow: Go to the previous sibling in a group of objects

Press 2 to go into the container. You should hear a pull-chain sound for a toggle button and "Read Message" as the label for the first toggle button. Practice using 4 and 6 to move between the toggle buttons. You should hear a "rebound" sound if you try to go past the four buttons in either direction.

Press 8 to go back up to the container and then press 6 to go to the next object which is a message bar. You should hear a printer sound and the the text of the message is read.

Press 6 to go to the next object, a text area. It should sound like a typewriter.

Press 6 to go to the next object. It's a container. You can go into it to get to the push buttons. The third push button "Undelete" should sound different since it is greyed out. (Greyed out means that this button is currently unavailable in the interface.) Since this a large group of push buttons, you can practice using two short-cuts to quickly move to the beginning or end of a list of objects. The controls are:

3/PgDn:	Move to the last object in a group
1/End:	Move to the first object in a group

We'll take a break in the tutorial for a few minutes as we describe all the sounds that you have been hearing. We'll pick the tutorial back up at this spot in the application interface when we finish the discussion on Sounds in UltraSonix.

Summary of controls presented in this section:

8/Up arrow:	Go up one level to the parent of the current object
2/Down arrow:	Go down one level to the first child of the current object
6/Right arrow:	Go to the next sibling in a group of objects
4/Left arrow:	Go to the previous sibling in a group of objects
3/PgDn:	Move to the last object in a group
1/End:	Move to the first object in a group

Summary of sounds presented in this section:

Tapping on glass:	Window
Opening door:	Container
Pull-chain:	Toggle button
Rebounding ball:	Out of bounds

Message bar:	Printer
Text area:	Typewriter
Keyboard tap:	Push button

4.6 Sounds in UltraSonix

UltraSonix uses combinations of synthesized speech and nonspeech audio sound effects to convey the state of a graphical application. The sounds effects are called auditory icons, like their graphical counterpart. The purpose of auditory icons is to convey the types of objects used in the graphical interface. You've already heard a number of sounds by now such as tapping on glass for a window and a typewriter for a text area. Other examples include:

Short pop:	Radio button
Short click:	Check box
Flipping shutter:	Menu
Single flip:	Menu button

A listing of all the sounds is provided in the reference section.

In UltraSonix, the sounds are also modified to convey attributes of the interface objects. For example, the push button "undelete" in the last example sounded muffled to convey that it was greyed out in the application interface. Conversely, a sound is exaggerated to indicate if an object is already selected or highlighted.

Auditory icons are also modified to convey the size of a object. In these cases, the pitch of a sound is modified. For example, the pitch of the typewriter sound indicates the relative size of a text area. A small text area (say 3 lines of text) would sound higher in pitch than a large text area (say 20 lines of text). This technique is also used with containers. A large container (contains 20 objects) would sound deeper in pitch than a small container (contains only three items.)

To hear examples of this technique, navigate back up to the container of push buttons by pressing the 8/Up key. You should hear a deep opening sound. By navigating to the next object (6/right key) you will move to a much smaller container. Likewise the next object is also a small container. Continuing through the objects (keep hitting the 6/right key), the next object is a small text area. The last object in this group is a large text area. Practice moving back and forth between these objects ending at the large text area that you started at. (This is the container with the push buttons "next", "delete", "undelete" and so on).

Summary of sounds presented in this section:

Short pop:	Radio button
Short click:	Check box
Flipping shutter:	Menu
Single flip:	Menu button
Muffled:	Greyed out object
Excited:	Selected or highlighted object
Deep pitch:	Large object (i.e. container or text)
High pitch:	Small object (i.e. container or text)

4.7 Advanced Navigation and Control

You should now be located at the large container that contains a number of push buttons. In this section, you are going to learn about pop-up windows, a new way to navigate between applications and a control for hearing the preview of a container. Also you are going to learn how to select or activate interface controls.

4.7.1 Pop-up dialogues and Selecting Objects

From the large container, enter the container by pressing the 2/Down key. Navigate through the list of push buttons until you hear the push button titled "Compose." (It's the 10th button in the group). So far you have been navigating through the application interface without actually operating any of the interface controls. It's important to be able to learn the contents of an interface without worrying about accidently pushing the wrong button and so on. For this reason, UltraSonix provides separate controls for navigation and for selection.

You may have heard sighted users talk about double-clicking on objects with their mouse in order to make something happen in an interface. In UltraSonix double-clicking is one form of selection. To select an object, you need to press the Enter key (on the numeric keypad) when you are located at that object. For example, you are currently located at the "Compose" push button. To select the push button, simply press the Enter key and you will hear a series of sounds.

The first sound you hear is a ripping sound. This is the sound for selection and indicates that you have selected the push button. In xmailtool, selecting the "Compose" button causes a pop-up dialogue to appear. A pop-up dialogue is another window that the application uses temporarily. In this application, the dialogue is used to write and send an electronic mail message. Right now, we're just going to practice creating and dismissing the pop-up dialogue.

The second sound you hear is a whistle sound with an rising pitch. This sound indicates that a pop-up has come into view. The last sound is a pogo-stick sound which indicates that you have been moved to the pop-up dialogue. If you press the 5 key, you will hear the pogo sound and "Pop-up dialogue."

The structure of the pop-up is pretty simple. Navigate down (2/Down arrow) to the window of the dialogue. Navigate down again and you will hear the container sound. Navigate to the right and you reach a text area. This is where you would write the email message. We'll explain how to do this in the next section. Navigate back to the container. In the container are two push buttons, reply and cancel. Move over to the cancel button and select it (Enter key). You will hear the ripping sound (selection) and a whistle sound with a descending pitch as the pop-up disappears. The last sound is the push button sound for the "Compose" button as you are moved to your last previous location before the pop-up. Practice selecting Compose and dismissing the pop-up with the Cancel button.

Summary of controls presented in this section:

Enter:	Selection

Summary of sounds presented in this section:

Ripping paper:	Selecting an object
Whistle up/down:	Pop-up appearing / disappearing

4.7.2 More on Moving Between Application

If you want to move back and forth between applications while maintaining your last known position you can use the controls shift-right and shift-left. For example, you should now be running two applications, the UltraSonix console and xmailtool. Move to the UltraSonix console by pressing and holding the shift key and then pressing the 4/left key, releasing both together. Then move back to xmailtool by pressing the shift and 6/right key combination. You should be now be located at your previous position, most likely the compose key.

Summary of controls presented in this section:

Shift-right:	Move to the next application, retain context
Shift-left:	Move to the previous application, retain context

4.7.3 Hearing a Preview of a Container

Sometimes you may want to get an overview of a portion of an interface without actually navigating throughout the interface contents. By pressing the combination shift-5, you can hear an auditory preview of any type of container. For example, navigate to the large container with all the push buttons. Press shift, holding it down, and then press 5 on the numeric keypad. You will hear short snippets of the auditory icons for all the objects in the container. In this case, you'll hear the beginning of the push button sound a number of times.

A window is simply a special case of a container. Press 8/Up to navigate to the main window of xmailtool and then press shift-5. You will hear short snippets of all the objects in the window. In this case, you will hear a number of different sounds since the window contains a number of different objects.

The purpose of a preview is to give you an overall feel for the contents of a container. You should be able to roughly gauge the number and diversity of objects in a container.

Summary of controls presented in this section:

Shift-5:	Hear preview of a container

4.7.4 Hearing the Object Hierarchy

In some cases, you may want to determine your current location by hearing the path from the top of the application tree to your current location. Like a preview, you will hear short auditory snippets for the objects in the path, from the top of the tree structure to your current location. To hear the path, simply press the alt key, holding the key down, and then press the 5 key on the numeric keypad. You may want to navigate to the Compose push button and practice requesting the path information.

Summary of controls presented in this section:

Alt-5:	Hear the path from the top to the current location

4.8 Accessing Text

In this section, we will discuss the different controls used to review and enter textual information. To practice this portion, you may want to navigate back to the UltraSonix console (Shift-left) and

then navigate to the text area. You will need to enter text mode to be able to read and enter text. You enter text mode by pressing the ./Del key on the numeric keypad. You will hear a rolling sound, like a drawer opening. This sound indicates entering and exiting text mode. You will not be able to enter text mode while on an object that does not support reading or writing text.

After you have entered text mode, type the following command: "more text-sample." This command will cause the contents of the file "text-sample" to be displayed on the console. You can use this sample, to learn how to review and enter text.

A common concept in screen readers is the distinction between the review cursor and the edit cursor. The review cursor indicates the current location for reading text while the edit cursor indicates the current location for editing text. UltraSonix provides controls for moving both cursors as well as controls for making the cursors point at the same location.

The basic commands are:

2/Down Arrow	Move down and read one line
4/Left Arrow	Move left and read one character
6/Right Arrow	Move right and read one character
8/Up Arrow	Move up and read one line
1/End:	Read this character
3/PgDn:	Read this word
5:	Read this line
7/Home:	Read this sentence
9/PgUp:	Read this paragraph

The data returned by these commands is passed through whatever filters are active for the current screenreader (see Controlling the Presentation of Text Using Filters, below).

The commands to read character/word/line/sentence/paragraph may be modified by the use of the shift, control, and meta keys. The "shift" key instructs the screenreader to read the next item, while the "control" key instructs the screenreader to read the previous one. For example, pressing the "control" and keypad 7 keys simultaneously instructs the screenreader to read the sentence previous to the current cursor position. Pressing the "shift", "meta", and keypad 3 keys simultaneously instructs the screenreader to read the next word from the current cursor position, and to update the current cursor ("move") to a position within that word. The shift and control keys may not be used simultaneously.

These commands move from the "active" cursor. The commands to control the various cursor modes are as follows:

/	Cursor status: announce the current cursor position, the current cursor (edit or review) and the following mode (see below)
shift + /	Toggle cursor mode: switch from edit to review mode or vice versa (see below)
control + /	Toggle follow mode (see below)

The "cursor mode" refers to the currently active cursor for the current screenreader. By default, each cursor begins in "edit" mode, which means that the cursor movement keys will control the edit cursor. Toggling the cursor mode will cause the screenreader to be placed in "review" mode, which will cause the cursor keys to control the review cursor instead. These cursors may be moved and queried independently of one another.

By default, in "edit" mode, the screenreader *reads* from the current position, outputting the requested information to the display device(s), but leaving the cursor in the same position. In "review" mode, by contrast, the screenreader *moves*, or outputs the requested information to the display device(s), and then updates the cursor to point to a location within the text that was just read. The "meta" key toggles the default read/move behavior for the current mode. That is, in "edit" mode the cursor is moved, and in "review" mode, the cursor reads without changing position.

"Follow mode" refers to the manner in which cursors behave when the cursor mode is toggled. In "following" mode, the review cursor will begin at the same location as the edit cursor when review mode is invoked. In "not following" mode, the review cursor maintains a persistent view of its own position; when review mode is invoked, the cursor will begin at the last location where it was before edit mode was invoked (of course, if review mode has not been previously invoked, the review cursor will begin start of the text area).

A useful control stops the playing of any speech and auditory icons that are currently being played. This control is the 0/Ins key at the bottom left of the numeric keypad.

O/Ins:	Stop all currently playing speech and audio

Some of the configuration for reading text is accomplished in the UltraSonix configuration files. See the next section for details.

4.8.1 Controlling the Presentation of Text Using Filters

Filters may be used to control the presentation of information to the user. The following sections describe the specification, configuration, and usage of the filtering system for the screenreader in UltraSonix.

4.8.1.1 What is a Filter?

A filter is an entity within the UltraSonix system which is defined with a series of commands which allow the user to alter the default presentation of text on the screen. A filter consists of one or more commands which are entered in a filter configuration file, where each command indicates a particular pattern in the text area which should be replaced by another. In this way, substitutions may be made from the standard way in which text is presented to the user to a user-defined presentation. A list of filters is loaded into each screenreader at startup. These filters are applied in the order that they are loaded.

A sample filter is described below:

```
# This is a comment and is ignored
filter test {
        "Bob"           "Robert"        # So is this -- everything until the
        "[Tt]he"        "that"          #  end of the line is tossed away.
        "[0-9]+"        "($old)"        # Enclose digits in parentheses
        HIGHLIGHT       "highlighting"
        FONT            "font"
        }
```

We will now explain, line by line, the format of the filters in the definition file. First, each filter begins with the keyword "filter", followed by a unique, case-sensitive name, and a pair of enclosing braces ("{ }"). The word filter is not case-sensitive, and may therefore be specified as "filter", "Filter", "FiLtEr", etc. A duplicate filter entry will discard earlier filter entries.

Following the initial opening brace are filter commands, specified one per line. Each filter command consists of a trigger and a replacement expression. The trigger is the condition which activates the filter, while the replacement expression is what the filter does when it is activated. We'll see exactly what this means as we examine each command.

The first command consists of the activation "Bob" and the replacement "Robert". This means that any instance of the characters "Bob" (the activation is case sensitive) will be replaced by the word Robert. For example, in the sentence: "Bob Bobbit went bowling", the filter would present the sentence "Robert Robertbit went bowling". The quotes around both the activation and replacement strings are only used by the filter to recognize phrases, and are not used when the filter searches for an activating pattern, nor when the pattern is replaced. To actually use quotation marks in either the activation string, the sequence \" must be used instead of the " character by itself. Whitespace and any characters following the # character are discarded by the filter parser, so you may feel free to indent, format, and comment your filter files as you see fit.

The second command introduces *regular expression* notation. More help on regular expressions may be found in the man pages for the "grep" utility, or in the lexx/yacc programming guide. Any activation enclosed in double-quotes is actually a *regular expression* and may use the standard regular expression notation to declare *sets* of patterns, rather than direct match patterns, such as "Bob" in the first command. This command actually searches for patterns which begin with either "The" or "the", and replaces it with "that".

The third command introduces the *expression substitution* variable. At times, it will be convenient to include the text of the pattern which triggered the activation in the replacement expression itself. This is done using the sequence of symbols "$old". When this is used, the pattern that matched the activation will be substituted into the position marked by $old in the replacement expression. For this command, any non-zero sequence of digits is found, and is surrounded by parentheses. For example, the sentence "Jan lives at 347 Huntsville Lane, Tacoma WA 50487" becomes "Jan lives at (347) Huntsville Lane, Tacoma WA (50487)". Note that the first time the activation pattern is triggered, the value of $old is 347, while the second time it is triggered, the value is 50487.

The last two expressions merely demonstrate activation patterns we have not seen before. Instead of regular expressions, which examine the content of the text presented to the user, the activation patterns HIGHLIGHT and FONT examine the actual onscreen presentation of the text itself. As the filter scans the text from that it is reading, it searches for changes in the text attributes stores in the representation model and inserts the appropriate replacement expression wherever the attribute change takes place.

The filter definition is terminated by a closing brace.

4.8.1.2 How are Filters Loaded?

Filters are defined in simple text files which have one or more declarations for filters. A declaration consists of the keyword "filter" and a unique name used to identify that filter, followed by the commands that the filter is to invoke. The names of the filters are case-sensitive, so the names "BOB" and "Bob", for example, refer to different filters. When UltraSonix is started up, each of the filter files defined in the variable "srFilterDefinitionFiles" is loaded, one at a time, in the

order they are specified. Each file then loads the individual filter definitions within the file in the order that they occur. If no errors occur in the parsing of the filter, its definition is stored in a dictionary of filter definitions, and may be referenced using the name specified in its definition.

If the configuration variable "srFilterDefinitionFiles" does not exist, only the filters in the file "default.flt" will be loaded.

Once filters have been loaded into the dictionary for the UltraSonix system, they may be registered for use with the screenreader. The filters to be used are specified in the variable "srFilters" in the order in which they are to be applied.

If the configuration variable "srFilters" does not exist, then the filters "SpeakCaps" and "SpeakUNIX" will be applied, in that order.

4.8.1.3 Using Filters in UltraSonix

The following section describes how to examine which filters are operative for a particular screenreader, as well as how to interactively enable and disable them.

*	Display current active status of filters.
Shift-*	Toggle filter enable mode. This feature may only be activated when UltraSonix is in text mode.

The filter display command causes each filter to be displayed to the output device. The position of the filter is announced, along with the name of the filter, and its current status (off or on).

Entering filter toggle mode allows the user to interactively enable or disable the current filters loaded using the keypad + and - keys. In filter toggle mode, UltraSonix cycles through the filters registered with the screenreader (specified in the "srFilters" configuration variable) and waits for the user to enable or disable each filter, using the keypad "+" and "-" keys. Pressing the "*" key announces the current filter by name, position, and current status (ON or OFF). Pressing the "+" key will enable the current filter, while pressing the "-" key will disable it. Filter toggle mode automatically exits when all filters have been configured in this manner, or the user may exit the mode manually by pressing Shift-*.

5.0 BRAILLE OUTPUT

5.1 Configuring UltraSonix to Use a Braille Terminal

UltraSonix supports the Alva ATB 3/20 and the Alva ATB 3/80 braille terminals. To use either Braille terminal with UltraSonix, connect the Braille terminals to a machine as specified by the user's manual. Then either start the server daemon, alvad from the command line, or specify the server in the config file (see the documentation on starting servers from UltraSonix).

The variable "brailleLoadable" must also be set to "Alva" in the config file (see Section 6.0, "Configuration," for more details):

```
brailleLoadable          = "Alva"
```

and the model must also be specified with the "alvaModel" variable:

```
alvaModel          = "Alva 3/20"
```

or

alvaModel = "Alva 3/80"

(3/20 is the portable 20 cell model with three status cells while the 3/80 is the 80 cell model with five status cells.)

These will tell UltraSonix to look for the loadable object named Alva.so in one of the directories specified by the loadableSearchPath variable in the configuration file.

5.2 Using the Alva Braille Terminal

Currently when the user is in text mode, the current line, word, or character, depending on the navigational mode, is sent to the Braille terminal as well as the speech device. The user can also navigate in the text area using the keys on the front panels of the Braille terminal. These keys work as the navigation keys when the user is in text mode.

In addition, there is a jumpscroll mode which will cause the Braille terminal to scroll to the next 20 or 80 characters on the line, depending on the model. To toggle in and out of this mode, use the command:

braille toggle jumpScroll

at the UltraSonix command prompt, or you can use the graphical console application to configure jump scroll mode.

There is also a command to retrieve the capabilities of the current Alva model:

braille cap <capability>

The capabilities the user can query are:

displayCells
statusCells
highlightSupported
hasCursorKeys
hasProgKey
hasHomeKey
otherKeys

6.0 CONFIGURATION

In order to use UltraSonix, the software must be configured properly. The UltraSonix software is higly configurable: it is possible to completely change the operation of the software by editing a set of configuration files. Most of the time, only very simple configuration is required: telling the system what hardware devices are attached, for example. At other times, "power users" may wish to fundamentally alter the behavior of the system.

Section 6.1, "Basic Configuration," describes the day-to-day configuration of the system, the directory layout and how it affects the operation of the software, and the tasks that must be performed to allow the software to run after installation. Section 6.2, "Advanced Configuration,"

describes more powerful configuration techniques that can be used to change the non-visual interfaces produced by UltraSonix.

6.1 Basic Configuration

This section describes the basics of how UltraSonix finds its required support files, how applications use the modified X libraries that are required to run under UltraSonix, and required modifications to the configuration file.

6.1.1 Essentials

UltraSonix should be installed on your system, preferrably in the directory /opt/GTsonicx. The GTsonicx directory contains the executables for UltraSonix and other programs, libraries used by these programs, and various configuration files.

Several of the subdirectories under GTsonicx are particularly important for configuration of the system:

6.1.1.1 lib

The lib directory contains libraries which are used by applications running under UltraSonix, and by UltraSonix itself. For applications to communicate with UltraSonix, they must use special, modified versions of the X Window System libraries that know how to communicate with the access software. Typically the substitution of these libraries is done at runtime by changing the LD_LIBRARY_PATH environment variable to "point" to the modified versions of the X libraries.

The easiest way to run applications under UltraSonix is to use the "sxrun" script. sxrun executes applications with the LD_LIBRARY_PATH variable set to use the modified libraries.

Only two X libraries have been modified to work with UltraSonix: libX11 and libXt. If any other X libraries are used by applications (libXm, libXmu, etc.), the unmodified versions of these can be used. The modified libX11 and libXt are *required* for applications to work however.

If you must set the LD_LIBRARY_PATH "by hand," or cannot use sxrun for whatever reason, you need to know how the libraries under the lib directory are used.

The subdirectories under lib are R5, R6, RAP, and R6.orig.

6.1.1.1.1 R6.orig

The R6.orig directory contains unmodified versions of the X11R6 libraries. These libraries should *only* be used by the UltraSonix executable itself. If applications use these unmodified libraries, they will be unable to communicate with UltraSonix since the required modifications are not present in these libraries.

6.1.1.1.2 R5

The R5 directory contains versions of the X11R5 libraries which have been modified to communicate with UltraSonix. libX11 and libXt have these changes; the other libraries in this directory are links to unmodified R5 libraries in /usr/openwin/lib.

Most applications should be run with their LD_LIBRARY_PATH set to this R5 directory. The R5 libraries are what we use in our test environment.

6.1.1.1.3 R6

The R6 directory contains modified libX11 and libXt versions from the X11R6 distribution. The other libraries in this directory are links to the unmodified R6 libraries in R6.orig.

There are currently very few R6 applications in existence; in fact current versions of Motif only support R5. We provide these R6 libraries as a "work in progress."

6.1.1.2 etc

The /etc directory contains various configuration files and scripts used by the UltraSonix software. It is essential that the UltraSonix software be able to find the contents of this directory when it starts. See the section below for a description of what these files do and how UltraSonix locates them at start-up time.

6.1.2 Finding the Configuration File

UltraSonix must be able to load a configuration file at startup time in order to operate. There are several places where the system looks for this file, and you can override the default locations in several ways.

By default, UltraSonix looks in order in the following locations for a configuration file:

$HOME/.mercator.config
~/.mercator.config
/opt/GTsonicx/etc/mercator.config

If a configuration file is not found in any of these locations, and an explicit path to it has not been provided, then the system will not start and an error message will be printed.

There are two ways to provide an explicit path to a configuration file. The first is to use the -f option on the command line when starting the system. The second is to set the MERCATOR_CONFIG environment variable to contain the path to the file. The -f option takes precedence over the environment variable.

6.1.3 Editing the Configuration File

The configuration file contains several options that describe how the system will behave. In particular, the file contains path information to various template files, filters, and sound files which the system must be able to locate to function properly. If you put an inappropriate option or wrong path into the configuration file, it is likely that UltraSonix will not function.

After installing the software, you must at a minimum tell the system what hardware devices are attached.

There are several "sections" in the configuration file that you can edit. The basic configuration options are described here; others are described in Section 6.2, "Advanced Configuration."

6.1.3.1 Paths

The config file specifies several paths to files which are needed by the system. In general, these paths *must* be set correctly or the system will fail.

The first path specifies the location of the various TCL scripts used by UltraSonix to build its interfaces:

```
mercatorTclPath          = "/opt/GTsonicx/lib/scripts"
```

The next important path specifies the location of the "template" files which govern per-widget and per-appliation behavior. This attribute specifies a list of subdirectories under which template files will be loaded. All files ending with the .tmpl suffix under these directories will be loaded:

```
templateDirectories      = ( "/opt/GTsonicx/lib/templates/Athena",
                             "/opt/GTsonicx/lib/templates/Motif",
                             "/opt/GTsonicx/lib/templates/Apps" )
```

The final path is a list of directories in which to search for loadable I/O drivers:

```
loadableSearchPath       = ( "/opt/GTsonicx/lib/loadables" )
```

6.1.3.3 Configuration of Loadable Modules

The configuration file specifies what loadable I/O drivers should be used by UltraSonix. Several attributes are used to specify which drivers should be loaded: speechLoadable, audioLoadable, and brailleLoadable tell UltraSonix which Speech, Audio, and Braille modules will be used. A fourth attribute, miscLoadables, contains a list of other modules to be loaded. Generally these are for input-only devices such as keypads:

```
speechLoadable           = "Dectalk"
audioLoadable            = "NetAudio"
brailleLoadable          = "Alva"
miscLoadables            = ( "Genovations" )
```

Note that the configuration above (Dectalk, NetAudio, etc.) represents the environment under which we run UltraSonix ourselves, so it is probably the most robust.

The loadable modules themselves may need to know various configuration information which they will try to retrieve from the config file. Check the docs for the particular I/O modules you are using to see what (if any) extra config information they may use. Below is the set of attributes used by the modules above, since they represent the default environment:

```
###
### Dectalk-specific configuration
###
dectalkServer            = ("/opt/GTsonicx/bin/dectalkd", "-f", "-t", "/dev/ttya")
dectalkTimeout           = 4

###
### NetAudio-specific configuration
###
netaudioServer           = ("/opt/GTsonicx/bin/netaudiod")
```

```
###
### Alva-specific configuration
###
alvaServer              = ("/opt/GTsonicx/bin/alvad", "-d", "/dev/ttyb")
alvaModel               = "Alva 3/20"
```

In general, particular I/O modules will look for information in the config file to start any servers that they may require, and set device-specific characteristics.

6.1.3.4 ScreenReader Configuration

The config file contains attributes which describe the default settings for the screenreader module. These are described in more detail in the screenreader documentation:

```
srOperationMode         = REVIEW
srSpeakingMode          = WORD
srNumberFormats         = WORD
srCapsMode                      = NODIFF
srTextControl           = FALSE
srMathControl           = FALSE
srMiscControl           = FALSE
srUnixSpeak                     = FALSE
srIncomingMode          = WORD
srSpeechRate            = 180
srVoice                 = "p\n"
srReturnClick           = "/opt/GTsonicx/sounds/TypeReturn.au"
srSpaceClick            = "/opt/GTsonicx/sounds/TypeSpace.au"
srKeyClick              = "/opt/GTsonicx/sounds/TypeKey.au"
srWrapSound                     = "/opt/GTsonicx/sounds/beep_jazz_piano.au"
```

The most important screenreader configuration options are used to control filter behavior. The srFilterDefinitionFiles is a list of files which contain filter definitions. The paths in this list must be accurate for the filters to be found. The srFilters attribute is a list of the filters to be used in each screenreader.

```
srFilterDefinitionFiles = ( "default.flt" )
srFilters               = ( "SpeakCaps", "SpeakUNIX" )
```

6.1.3.5 Output Configuration

You can control the amount of output generated by UltraSonix, as well as where that output should be sent:

```
errorLevel              = 4
errorOutput             = STDERR
```

ErrorLevel is a "filter" which controls how much output will pass the system. The value of 8 allows all output to be written. 4 is useful as it only allows severe warnings and errors to be printed.

6.1.3.6 Console Configuration

The config file specifies what (if any) console application will be run when UltraSonix starts.

```
console          = ("/opt/GTsonicx/gui-console")
consoleWait      = 5
consoleEnv       = ( "LD_LIBRARY_PATH=/opt/GTsonicx/lib/R5" )
```

The "console" attribute provides the path of the console application to be started automatically. "ConsoleWait" specifies how long UltraSonix will wait for the console to start, before it defaults to using standard input and output for I/O. The "consoleEnv" attribute is used to provide an alternative environment for the console process. Here we specify the path to our modified X libraries so that the console will be available under UltraSonix.

6.2 Advanced Configuration

UltraSonix provides extensive support for customization by administrators and "power users." This section describes some of these facilities. Section 6.2.1 describes the notion of templates, which are used to customize behavior of objects in UltraSonix. Section 6.2.2 details some additional features in the configuration file that can be used for advanced customization. Section 6.2.3 describes how to extend the configuration file and template file parsers to accept new attributes without the need to recompile the system. Section 6.2.4 describes the tcl files that UltraSonix uses to build its non-visual interfaces.

6.2.1 Template Configuration

UltraSonix uses the notion of "templates" to control the presentation and use of particular widgets and applications. By editing a template file, you can change how the system will respond when it encounters a particular object in an application. Templates capture the widget-and application-specific information required by UltraSonix.

6.2.1.1 Widget Template Configuration

Each class of objects (otherwise known as widgets) can be configured using templates. Here is the template for the Motif widget XmPushButton.

```
classTemplate XmPushButton {
  navigable         = TRUE
  sound             = "Sounds/winding.au"
  sensitive         = [SensitiveProc]
  mappedWhenManaged  = [MapWhenMgdProc]
}
```

The token "classTemplate" is a reserved word in the template parser that indicates that this template is specifying the attributes of an entire class of widgets. The string "XmPushButton" denotes the widget class that this template will be applied to.

Within the braces are a set of attributes that are being assigned to this template. Each attribute understood by UltraSonix has a fixed type that it can take; allowable types are boolean, string, integer, and list of strings. Further, the value of each attribute can specified either "simply" (by providing an integer, string, and so forth), or by a "procedure" (see below).

Four attributes are shown here. The "navigable" attribute is of type boolean, and indicates whether objects of this class will be considered "viewable" to UltraSonix. The "sound" attribute indicates the sound to play when the user navigates to an object of this class, and is of type string. The next two attributes, "sensitive" and "mappedWhenManaged" are of type boolean, but their values are specified using *procedures*.

The square bracket notation in a template file indicates that the string inside the bracket is the name of a tcl procedure that should be run whenever the value is needed. The use of procedures allows dynamic setting of template attributes. (TCL stands for Tool Command Language, which is an interpreted, interactive language). The tcl code for SensitiveProc is shown below:

```
proc SensitiveProc {type node} {
   if { [resource $node get sensitive] == "TRUE" } {
      return "TRUE"
   } else {
      return "FALSE"
   }
}
```

Writing these procedures will require an understanding of tcl and, at least, a partial understanding of UltraSonix internals. Many users will not need to configure UltraSonix at this level.

By using class templates, you can affect change in the way UltraSonix handles entire classes of widgets: the XmPushButton example shown above will change the behavior for all Motif Push Buttons in all applications running under UltraSonix.

Many times it may be useful to selectively change the behavior of one particular widget in one particular application, however. To this end, UltraSonix supports the notion of "object templates." Object templates work exactly the same as class templates, only they allow you to uniquely specify one individual widget to which the specified attributes will apply.

Here is an example of an object template that overrides the default push button behavior for one particular push button, this one in xmailtool:

```
objectTemplate XMailTo6l.outer_box.quit {
   sound            = "Sounds/flush.au"
}
```

Note here the use of the key word "objectTemplate" (rather than "classTemplate") to indicate that we're specifying one particular object, rather than an entire class of objects. Also, the specification of the object we're interested in is the "long name" of the widget. This long name is similar to Xrm-style naming in the X Window System, except that wildcards are not supported, and the specification tokens must be widget names, not classes.

Both object and class templates support exactly the same attributes; they just provide different mechanisms for setting those attributes. Any values not explicitly provided in an object template will "fall back" to the class template for the widget class.

6.2.1.2 Application Template Configuration

UltraSonix also provides the notion of "application templates" to control per-application settings. Currently, application templates are not widely used. Here is an example of a simple application template, however:

```
appTemplate XMailTool {
  readySound    = "/opt/GTsonicx/lib/Sounds/electric.au"
  shutdownSound = "/opt/GTsonicx/lib/Sounds/flush.au"
}
```

This example shows two attributes that control the start up and shut down sounds that will be played for xmailtool. Note that application templates support a *different* set of attributes than class/object templates. Application template attributes specify behavior for an entire application, not just one widget or class of widgets.

6.2.2 Advanced Configuration File Settings

As we have seen, templates provide a tool for customizing the behavior of widgets, classes of widgets, and applications. To simplify the writing of templates, UltraSonix allows you to omit certain attributes when you write a new template file. For example, you may wish to create a class template for a new widget class, "XmFooBar," but have most of the attributes be the same as other class templates.

Thus, to simplify the creation of templates, and to standardize behavior, UltraSonix allows you to specify "fallback defaults" for template attributes in the configuration file. These fallbacks are the values that your template attributes will take if values for them are not explicitly provided.

As a safety mechanism, UltraSonix *requires* you to specify fallbacks for all template attributes in your configuration file. This prevents the case where you omit an attribute from a template and also neglect to provide a fallback for it (which would cause a runtime error).

UltraSonix will inform you if you omit a required fallback value from the configuration file.

Be careful about changing these fallbacks; changing them carelessly may cause unexpected behavior:

```
defaultClassNavigable      = TRUE
defaultClassShell          = TRUE
defaultClassAllowTextMode      = FALSE
defaultClassSound          = "/opt/GTsonicx/lib/Sounds/bong.au"
defaultClassSpeakOnEnter = ""
defaultClassSpeakOnInfo  = ""
```

Note that the names for the fallbacks are derived from the template attribute names.

6.2.3 Extending the Configuration File/Template File Parser

If you need to extensively customize the template mechanism used by UltraSonix, you may find that it would be helpful to add new attributes to templates. By default, the template file parser understands only a handful of "hard-coded" attributes that are already understood by the internals of UltraSonix.

There is a mechanism for extending the parser so that it can understand new attributes easily, however. You can extend the parser by editing the file "attributes" file. UltraSonix searches for an attributes file in much the same way it searches for a configuration file.

By default, UltraSonix looks in order in the following locations for an attribute file:

$HOME/.mercator.attrib
~/.mercator.attrib
/opt/GTsonicx/etc/mercator.attrib

If an attribute file is not found in any of these locations, and an explicit path to it has not been provided, then the system will assume that you're not augmenting the template parser with an attributes file.

There are two ways to provide an explicit path to an attributes file. The first is to use the -a option on the command line when starting the system. The second is to set the MERCATOR_ATTRIB environment variable to contain the path to the file. The -a option takes precedence over the environment variable.

Here is an example of an attributes file that extends the template parser to add several attributes (the attributes shown here are actually already used by UltraSonix, and should not be added to the attributes file):

```
# NAME       TYPE     FALLBACK NAME          TEMPLATE TYPE
#########################################################################
navigable    BOOLEAN  defaultClassNavigable      CLASS
volume       INTEGER  defaultClassVolume         CLASS
sound        STRING   defaultClassSound          CLASS
readySound   STRING   defaultAppReadySound       APP
```

The attributes file is based on a four-column format. The first column gives the name of the attribute that will now be viable in the template files. The second is the type of the attribute (either BOOLEAN, INTEGER, STRING, and STRINGLIST. The third column provides the name that will be used as a fallback in the configuration file (recall that all template attributes *must* have a fallback value specified in the configuration file). The fourth column indicates what type of templates the new attribute may appear in. Legal values are CLASS (indicating that the attribute can appear in either class or object templates), and APP (indicating that the attribute can appear in application templates).

Remember that if you add a new attribute via the attributes file, then you must provide a fallback value for it in the configuration file. UltraSonix will not start if these two files are "out of sync."

6.2.4 Using TCL Files

The UltraSonix software contains an embedded interpreter for the tcl language. All interfaces in UltraSonix are generated through tcl scripts that are run in response to events: either changes in the application state or input by users. By expressing interface behavior in tcl, separately from the "core" system, which is implemented in C++, we can easily change the behavior of the system to support radically new interfaces.

Tcl scripts specify the following:

- How to handle user input
  "What happens when the user pressed the 5 key on the keypad?"
- What interface output to generate
  "What output is presented when the user navigates to a grayed out pushbutton?"
- When to simulate application input
  "What happens when the user presses the enter key?"
- Querying the interface model
  "Is the current object sensitive to user input?"
- How to handle dynamic application interface changes
  "What happens when a dialog box appears?"

When UltraSonix first starts, it loads a tcl script that installs a set of behaviors (called "actions") that are executed whenever an application's interface changes, and a set of "bindings" that specify what will happen when the user generates some input.

As shipped, the first tcl file loaded into the system is called "mercator.tcl." Mercator.tcl establishes a basic set of actions and bindings, and defines some utility procedures. It also loads the following other tcl files:

text.tcl	Provides text-mode support.
navigate.tcl	Basic navigation algorithms.
audio.tcl	Audio support for playing sounds.
templateprocs.tcl	Procedures used in templates.

Any of these files may be changed to extend or alter the behavior of the system.

7.0 REFERENCE

7.1 Navigation Commands

8/Up arrow	Go up one level to the parent of the current object
2/Down arrow:	Go down one level to the first child of the current object
6/Right arrow:	Go to the next sibling in a group of objects
4/Left arrow:	Go to the previous sibling in a group of objects
3/PgDn:	Move to the last object in a group
1/End:	Move to the first object in a group
Alt-right:	Move to the next application
Alt-left:	Move to the previous application
Shift-right:	Move to the next application, retain context
Shift-left:	Move to the previous application, retain context
Shift-5:	Hear preview of a container
Alt-5:	Hear the path from the top to the current location
Enter:	Selection
+:	Begin/End Drag/Release
./Del:	Enter/Exit text mode

7.2 Sounds

Tapping on glass:	Window
Opening door:	Container
Message bar:	Printer
Text area:	Typewriter
Keyboard tap:	Push button
Pull-chain:	Toggle button
Rebounding ball:	Out of bounds
Text area:	Typewriter
Winding:	Application starting
Flushing:	Application ending
Paper shuffle:	Switching between applications
Music:	Focus moved to an application
Ripping paper:	Selecting an object
Whistle up/down:	Pop-up appearing / disappearing
Muffled:	Greyed out object
Excited:	Selected or highlighted object
Deep pitch:	Large object (i.e. container or text)
High pitch:	Small object (i.e. container or text)

7.3 Text Review and Editing Commands

./Del:	Enter/Exit text mode
O/Ins:	Stop all currently playing speech and audio
1/End:	Read this character
3/PgDn:	Read this word
5:	Read this line
7/Home:	Read this sentence
9/PgUp:	Read this paragraph
Shift-{1,3,5,7,9}	Read next item
Control-{1,3,5,7,9}	Read previous item
Meta-{1,3,5,7,9}	Toggle default move/read behavior
/	Announce cursor
Shift-/	Toggle cursor mode
Control-/	Toggle follow mode
*	Announce filter status
Shift-*	Toggle filter enable mode
+	Enable filter (filter enable mode only)
-	Disable filter (filter enable mode only)

7.4 Class and Object Template Attributes

This section describes the attributes that are supported in class and object templates. Recall that both object and class attributes support the same attributes; they merely provide different mechanisms for naming the objects those attributes will be associated with.

Each entry here lists the name of the attribute that appears in the template, the type of the attribute, and the name of the fallback for the attribute, which appears in the configuration file.

navigable BOOLEAN defaultClassNavigable

Indicates whether or not the object will be considered "navigable" by UltraSonix. Classes that are not navigable are essentially "invisible" and are ignored by the system.

shell BOOLEAN defaultClassShell

Indicates whether or not the object will be treated as a shell (top-level window). The fallback should be set to TRUE and overridden for non-shell classes. This is so application shells (which are created with a class name different than ApplicationShell) will be correctly detected as top-level windows.

allowTextMode BOOLEAN defaultClassAllowTextMode

If set to TRUE, allowTextMode indicates that the object will support text mode navigation.

sensitive BOOLEAN defaultClassSensitive

The sensitive attribute is used to indicate whether or not an object is sensitive (that is, whether it will accept user input). This is typically set to a procedure that retrieves the value of the sensitive resource from the object.

looped BOOLEAN defaultClassLooped

Indicates whether the "entry" sound for this object will be looped (played repeatedly).

mappedWhenManaged BOOLEAN defaultClassMappedWhenManaged

Determines whether objects that are managed will also be mapped. This attribute is typically set to a procedure that retrieves the value of the mappedWhenManaged resource from the widget.

allowEscape BOOLEAN defaultClassAllowEscape

Indicates whether it is possible for users to exit dialog boxes by navigating up and out of them (if set to TRUE), or whether users are confined to dialog boxes as long as they are posted (if set to FALSE).

volume INTEGER defaultClassVolume

The volume at which to play the "entry" sound for this object.

muffle	INTEGER	defaultClassMuffle

The amount of muffling to use when playing the "entry" sound for this object.

rate	INTEGER	defaultClassRate

The rate of speed at which to play the "entry" sound for this object.

delay	INTEGER	defaultClassDelay

The amount of delay before playing the "entry" sound for this object.

leftMargin	INTEGER	defaultClassLeftMargin
rightMargin	INTEGER	defaultClassRightMargin
topMargin	INTEGER	defaultClassTopMargin
bottomMargin	INTEGER	defaultClassBottomMargin

The values of the margins for this object. These attributes are only used by the ProtoTextRep class; typically they are set to procedures that retrieve the appropriate resources from the widget.

sound	STRING	defaultClassSound

The "entry" sound for this object.

speakOnEnter	STRING	defaultClassSpeakOnEnter

The string to speak when this object is entered.

speakOnInfo	STRING	defaultClassSpeakOnInfo

A string of "extra" text that is spoken when the Info action is invoked on this object.

unsafeResources	STRINGLIST	defaultClassUnsafeResources

A list of the resources that may be set by the application or widget in a way that bypasses the RAP hooks. Unsafe resources must be retrieved explicitly each time they are used, and are thus extremely expensive. (NOTE: unsafeResources is not currently implemented.)

creation	INTEGER	defaultClassCreation
destruction	INTEGER	defaultClassDestruction
mapped	INTEGER	defaultClassMapped
unmapped	INTEGER	defaultClassUnmapped
managed	INTEGER	defaultClassManaged

unmanaged	INTEGER	defaultClassUnmanaged
realized	INTEGER	defaultClassRealized
unrealized	INTEGER	defaultClassUnrealized

The values of these attributes are retrieved each time an object is created, destroyed, and so on. They are typically set to procedures that invoke some widget-instance or widget-class specific functionality. In essence, they provide a widget-specific variant of the Action mechanism.

srFilters STRINGLIST defaultClassSrFilters

A list of the filters that should be installed on this object's screen reader.

srDelimitChars STRING defaultClassSrDelimitChars

A string of the characters that are used as delimiters by the object's screen reader.

srTerminalChars STRING defaultClassSrTerminalChars

A string of the characters that are used as terminals by the object's screen reader.

7.5 Application Template Attributes

readySound STRING defaultAppReadySound

The sound to play when this application becomes ready. (NOTE: This attribute is currently not implemented.)

shutdownSound STRING defaultAppShutdownSound

The sound to play when this application shuts down. (NOTE: This attribute is currently not implemented.)

blockCursor BOOLEAN defaultAppBlockCursor

Indicates whether the block-cursor detection code should be used in this application. The block-cursor detection code is used to track the location of the application cursor in text areas.

7.6 Actions

Actions are call-out points that are invoked when the UltraSonix off-screen model changes. Arbitrary TCL code can be associated with actions, and will be invoked whenever the action is evaluted.

This section describes the actions built in to UltraSonix, when they are called, and the arguments that are passed to the TCL procedures associated with them.

BrailleProg

Invoked whenever the "prog" button on the braille keyboard is pressed.

BrailleHome

Invoked whenever the "home" button on the braille keyboard is pressed.

BrailleCursor

Invoked whenever a cursor button on the braille keyboard is pressed.

BrailleUpBounds

Invoked whenever the up key is pressed on the braille keyboard, and the new position would pass beyond the top of the data buffered in the braille loadable module.

BrailleUp

Invoked whenever the up button is pressed on the braille keyboard.

BrailleDownBounds

Invoked whenever the down key is pressed on the braille keyboard, and the new position would pass beyond the bottom of the data buffered in the braille loadable module.

BrailleDown

Invoked whenever the down button on the braille keyboard is pressed.

BrailleLeftBounds

Invoked whenever the left key is pressed on the braille keyboard, and the new position would pass beyond the edge of the data buffered in the braille loadable module.

BrailleLeftJumpScroll

Invoked whenever the braille display jump scrolls left.

BrailleLeft

Invoked whenever the left button on the braille keyboard is pressed.

BrailleRightBounds

Invoked whenever the right key is pressed on the braille keyboard, and the new position would pass beyond the edge of the data buffered in the braille loadable module.

BrailleRightJumpScroll

Invoked whenever the braille display jump scrolls right.

BrailleRight

Invoked whenever the right button on the braille keyboard is pressed.

ClientCreated client name, client ID

Invoked whenever a new client has been detected but is not yet ready.

ClientDeletion client ID

Invoked whenever a client is destroyed.

GoTo obj ID

Invoked throughout UltraSonix whenever the user's current position must be changed.

ClientReady client ID

Invoked whenever a new client has become ready for use.

ClientShutdown client ID

Invoked whenever the client shutdown process is initiated.

StopSpeaking

Generated from an external keypad when the 0 key is pressed.

ReadThisChar

Generated from an external keypad when the 1 key is pressed.

DownPressed

Generated from an external keypad when the 2 key is pressed.

ReadThisWord

Generated from an external keypad when the 3 key is pressed.

LeftPressed

Generated from an external keypad when the 4 key is pressed.

FivePressed

Generated from an external keypad when the 5 key is pressed.

RightPressed

Generated from an external keypad when the 6 key is pressed.

ReadThisSentence

Generated from an external keypad when the 7 or 9 key is pressed.

UpPressed

Generated from an external keypad when the 8 key is pressed.

ChangeTextMode

Generated from an external keypad when the . key is pressed.

SelCurrent

Generated from an external keypad when the enter key is pressed.

PopupReturn

Invoked whenever the a popup window is dismissed.

EnterNotify obj ID

Invoked whenever the pointer enters a new object.

ButtonPress obj ID, button state, client ID

Invoked whenever a button press is detected.

ShellMapped obj ID, client ID

Invoked whenever a new top-level window is mapped.

MapNotify obj ID, client ID

Invoked whenever any non-shell object is mapped.

XtObjectCreation obj ID, parent ID, obj class

Invoked whenever any object is creatd.

XtObjectDestroyed obj ID

Invoked whenever any object is destroyed.

XtObjectChange old obj name, old obj ID, new obj ID

Invoked whenever the current location changes.

XtObjectUnmapped obj ID, client ID

Invoked whenever an object is unmapped.

XtObjectRealized obj ID

Invoked whenever an object is realized.

XtObjectUnrealized obj ID

Invoked whenever an object is unrealized.

XtObjectManaged obj ID

Invoked whenever an object is managed.

XtObjectUnmanaged obj ID

Invoked whenever an object is unmanaged.

ULTRASONIX DESIGN DOCUMENT

1.0 INTRODUCTION

This document contains details about the implementation of various aspects of UltraSonix. The goal of this document is to provide enough information about key UltraSonix subsystems that future developers will have a "head start" on understanding and maintaining the code.

Most of the sections of this document were originally written as "stand-alone" design documents during the course of development. We have tried to keep the information here as up-to-date as possible but, of course, the source code is the final arbiter of the implementation.

2.0 HANDLING INPUT

2.1 Basic Concepts

All input to UltraSonix is done through the FDInterest class. If you want to use a new input stream (for example, a new device which provides input to the system), you create a new subclass of FDInterest. The FDInterest base class tracks all of the instances of its subclasses, and remember which file descriptors they are interested in (hence the name "FDInterest").

UltraSonix's central loop is a call to FDInterest::MainLoop(). This code blocks in poll(), waiting for activity to occur on any of the descriptors it is looking at. When activitiy occurs (for example, data is available for reading), MainLoop() determines the specific FDInterest subclass which has expressed an interest in that descriptor, and calls the HandleActivity() method on it.

2.2 Adding New FDInterest Subclasses

Each subclass of FDInterest has the following responsibilities:

1. Tell the FDInterest class which descriptors it will be responsible for.

In the constructor for the subclass of FDInterest, you should open any descriptors that you will be responsible for (sockets, files, devices, etc.). You then need to tell the FDInterest base class about these descriptors, so that it can add them to its poll set. You do this by first calling

```
AddFDInterest(fd);
```

to create an association between your instance and the descriptor, and then by calling

```
SetIOMask(fd, MASK);
```

to tell the FDInterest class what types of activity you will handle. Most of the time you are interested when data is available for reading, so you can pass the symbol "ARead" as the MASK parameter to SetIOMask(). Look at FDInterest.h for other mask symbols.

Note that you can later change the types of activity you are interested in via other calls to SetIOMask() (for example, to be notified whenever a descriptor is writable or has an exceptional condition). You can also be "interested in" multiple descriptors, although no two FDInterest subclasses can be interested in the same descriptor.

2. Implement a HandleActivity method to deal with data available on its descriptors.

Your subclass MUST implement a HandleActivity() method which will be automatically called whenever any of the descriptors you have specified an interest in (via AddFDInterest()) have activity on them that matches the activities you said you would handle (via SetIOMask()).

HandleActivity() is called with two arguments: the descriptor and the type of activity which has been seen on it (for example, readability or writability). HandleActivity() should return a non-zero value if some error occurs.

This is the function where the "smarts" of your FDInterest subclass reside. Only your class can know how to interpret data from its file descriptor.

NOTE that even if you don't specify that you are interested in errors and exceptional conditions (via a call to SetIOMask()), HandleActivity() will still be called whenever these special conditions occur.

3. Implement the "shutdown" protocol required to cleanly terminate an FDInterest subclass.

There are two cases in which an FDInterest subclass may wish to delete itself. In the first, the subclass detects some exceptional condition which cannot allow it to continue (for example, if a subclass represents a connection to another process, and the process dies, the subclass may wish to be deleted). In the second, the user has requested that UltraSonix shut down completely and the system will message each subclass to shut themselves down in an orderly manner.

Since other components in UltraSonix may be dependent on your subclass, it is important that you follow the shutdown protocol outlined here. This protocol ensures that the shutdown of FDInterest subclasses is properly ordered and that all references to the subclass are cleaned up.

If a client wishes to shut itself down, it simply calls RemoveFDInterest(fd) to tell the FDInterest base class that it is no longer handling activity on the specified descriptor. Next, it calls MarkForCleanup() to tell FDInterest that it is ready to be deleted. FDInterest will actually delete the subclass at some point in the future, after all necessary housekeeping has been done.

When UltraSonix wishes to shut down an FDInterest subclass, it calls the Shutdown() method on that subclass. All classes derived from FDInterest should implement Shutdown() and perform any class-specific shutdown activities here. Before returning, Shutdown() should call MarkForCleanup() to ensure that the FDInterest class will actually be deleted.

3.0 INTERFACE MODELING

3.1 Introduction

UltraSonix keeps a model of the user's "desktop" environment as it runs: all of the applications currently executing. This model is stored as an in-memory collection of objects, representing applications, widgets and gadgets, and resources within widgets and gadgets.

This model can be queried to retrieve information about graphical interfaces. It is automatically updated via the RAP protocol as applications change state.

3.2 Fundamentals

The three most important classes in the off-screen model, the AppModelManager, the Client, and the XtObject, are all derived from class Storage. The Storage base class enforces an API for mapping from events to actions.

All Storage-derived classes maintain an "Event/Action" dictionary. UltraSonix allows the execution of arbitrary "actions" when certain events are received (this is how the system implements keybindings). The association between events and actions may be established globally, on a per-client basis, or a per-widget basis. Each subclass derived from Storage implements a different "level" of these mappings.

Note that all UltraSonix interface modeling data structures store pointers to the actual data objects. Thus, no copies are made (either in or out) when data is inserted or retrieved from the model.

Also note that both Clients and XtObjects have unique "names" (which are really strings) that are used to identify them. For XtObjects, these names have the form XtObjectXXX (where XXX is some unique number); for clients, these names have the form ClientXXX (again, where XXX is some unique number). These names are used throughout the system to uniquely identify a given object instance, and are visible through the TCL interfaces.

3.2 The Application Model Manager

The Application Model Manager (AMM) is the outermost "entry point" into the off-screen model. When UltraSonix starts, it instantiates one global instance of an object of class Mercator, which holds all global data that is needed by the system. The Mercator object creates and holds a pointer to one AppModelManager instance, which stores the offscreen model. Thus, there is only one instance of this object, representing the entire desktop environment.

The AMM stores the following data representations:

clientList	The authoritative list of all client applications running on the desktop.
currentClient	A pointer to the current client (may be NULL if there is no current client).
clientNames	A mapping of client names to client instances. This data structure allows the lookup of clients based on their identifiers (names).
xtobjectNames	A global mapping of XtObject names to XtObject instances.
xtobjectWindows	A global mapping of XtObject windows to XtObject instances.
eventActions	The global map from event types to actions to be executed.

The AMM provides the "global" mappings that represent the user's universe. Thus, the AMM maintains a list of all clients, and a way to retrieve specific clients given their names.

Likewise, it also maintains global mappings of XtObjects (representations of widgets and gadgets). The "typical" way to access XtObjects is through the client that contains them, but often callers do not know a priori the client containing a given object (usually because the caller only knows the

identifier or window of the object). Thus, the AMM maintains mappings that index XtObjects by these attributes.

3.3 Representing Clients

Running applications are represented by Client instances. Client objects maintains all salient attributes of a client application:

clientName	The name of this client (as provided by the application writer).
clientClass	The class of this client (as provided by the application writer).
id	The server-supplied XID representing this client.
uniqueName	The string uniquely identifying this application.
fontCache	A cache of the fonts in use by this client.
xtObjectObjectIds	All of the XtObjects contained in this application, indexed by object IDs.
currentLocation	The current location within the client, stored as a pointer to an XtObject (or NULL if there is no current location).
topWindows	A list of all the "top-level" XtObjects comprised by this application.
eventActions	The client-scope mapping of events to actions.
ready	A boolean value, indicating whether or not the client is ready for interaction.

Most of the data stored by clients is self-explanatory. The fontCache member maintains a representation of the attributes of all fonts in use by this application. The topWindows member is a "short-cut" data structure for maintaining information about top-level windows. The ready member indicates whether a client has finished the RAP start-up phase (a new client instance is marked as not ready until UltraSonix receives information about its widget hierarchy and resources).

3.4 Representing Widgets and Gadgets

All widgets and gadgets are represented as instances of class XtObject. XtObjects store all information relevant to a particular widget. The data contained in XtObjects includes:

id	The Xt-internal identifier for this widget (guaranteed to be unique within a given client).
myName	The name for this widget, as supplied by the application writer.
myClass	The class for this widget, as supplied by the application writer.

longName	The Xrm-style (dot notation) name of this widget, suitable for use in .Xdefault files or in object templates.
uniqueName	The UltraSonix-internal string, uniquely identifying this instance.
xtObjectIdent	A numeric representation of the uniqueName string.
parent	A pointer to the XtObject that is the parent of this object (may be NULL if this object has no parent).
window	The X Window ID of this object's window.
x, y	The location of this object, relative to its parent.
width, height	The size of this object in pixels.
mapped	Boolean indicating map status.
managed	Boolean indicating managed status.
borderWidth	Width of the widget border, in pixels.
grabsInstalled	A book-keeping boolean, used to indicate whether or not we've already installed our needed key grabs on this object.
resources	A pointer to a ResourceCache instance, which stores all of the resources associated with this XtObject.
eventActions	The per-widget mapping of events to actions.
classDictionary	The per-widget class mapping of events to actions.
sreader	A pointer to the screen reader instance associated with this XtObject (if there is one).
textrep	A pointer to the text rep instance associated with this XtObject (if there is one).
client	The client containing this object.
children	A list of all children of this object.
popReturn	A bookkeeping variable, popReturn holds a pointer to the object previously containing focus before a popup appeared.

Most of these members are self-explanatory, but a few require a bit more exposition. The window member contains the 32-bit window ID of the widget's window. If the widget is not yet realized (that is, it has no window associated with it), the value will be RapUnrealized (#defined to be 0). If the widget is a gadget (windowless widget), the value will be RapGadget (#defined to be 2).

The resources member is described more fully below, under "Representing Resources."

The classDictionary member is a dictionary of pointers to event/action dictionaries. The key to this dictionary is a string, which represents the class names of widgets. The resulting value is a set of event-to-action bindings for widgets of this class. This dictionary, which is static (shared by all XtObject instances), is used to allow the association of events to actions on a per-widget class basis (as opposed to a per-widget instance basis).

If a given object supports text mode (as indicated by the "allowTextMode" attribute in its template), then a TextRep instance will be created in the textrep member. The first time a user actually enters text mode in this XtObject, a screenreader instance will be created (and stored) in the sreader member to provide an interface to the textrep.

The popReturn member is used to essentially create a popup return stack within the XtObject instances themselves. When a popup appears, UltraSonix warps the user to the popup, and stores the old current location in the popReturn member of the popup. When the popup disappears, the current location is set to the value of its popReturn. This scheme works for arbitrary depths of popups.

3.5 Representing Resources

The "outer" representation of resources is a class called ResourceCache, defined in Resource.h. An instance of a ResourceCache will be associated with every XtObject and used to store the resources contained by that object.

ResourceCache provides an interface to retrieving and setting resource values. The implementation of ResourceCache is completely hidden (and fairly complex). Internally, ResourceCache maintains dictionaries of Resource instances. Resource instances each represent one particular resource: its name, class, type, and value. Values are represented by the ResourceValue type, which supports arbitrary-sized data representations.

All type-specific information needed to represent a given resource value is maintained by an object called the ResourceTypeManager. The ResourceTypeManager provides a mapping between resource types ("XmString", for example) and information about the size of such representations, how to copy and print resources of such type, and so on.

At startup time, UltraSonix "registers" a handful of common types with the ResourceTypeManager. Registration creates an association between the resource type name and information about the size of the type and functions needed to print, store, copy, and free objects of the type.

UltraSonix ignores resources with types that have not been registered with the ResourceTypeManager, since it cannot know how to interpret the binary value data returned by RAP unless the type has been registered.

Registration for new types should be added to the function InitializeResourceTypeManager() in Resource.cc.

See the file Resource.cc for more information regarding how resources are stored and managed.

CAVEAT: The current resource code is specific to 32-bit machines. The code for decoding resource values is word-size dependent, and will have to be fixed to run on non-32-bit machines.

3.6 Miscellaneous

In addition to the high-level (Xt-based) representations of X interfaces, there are a number classes used to represent lower-level (X protocol-based) constructs. This section discusses two of those, used to store Graphics Contexts and Fonts.

3.6.1 Graphics Contexts

Graphics Contexts, or GCs, are structures that hold information used to parameterize a drawing operation. Examples of information contained in GCs include line width, color information, and font information. GCs are used in all X protocol requests involving on-screen rendering.

UltraSonix stores GC information so that it can determine the font being used to render text (for ProtoTextRep text modeling), and to capture attribute information about drawn text.

A class called GCValueDict is instantiated in each RAP object to maintain a mapping from GC IDs (32-bit numeric values) to instances of class GCValue. GCValue is the UltraSonix-internal representation of a Graphics Context.

Each GCValue maintains a Graphics Context structure, and supports comparison operations (for determining if the color of two adjacent pieces of text differ, for example). GCValues are primarily used by the ProtoTextRep class.

3.6.2 Fonts

Fonts information is stored in objects of class FontCache. FontCache maintains a dictionary of font attributes (represented as XFontStructs), indexed by X Font identifiers (which are 32-bit values).

Each client keeps its own FontCache, since the identifiers assigned to each font are particular to a given client (thus, even if two clients open the same font, the IDs they use to refer to these fonts will be different).

The FontCache class provides methods for adding, removing, and retrieving font attribute information from the cache. There are also a number of "short-cut" functions for determining bounding regions around strings of text, given a font ID that the text is rendered in. These methods are used extensively by the ProtoTextRep class.

Note that it is impossible to determine actual font names from font IDs, only font attributes such as size information.

4.0 INFORMATION RETRIEVAL

4.1 Introduction

UltraSonix retrieves information about graphical interfaces from running applications. Applications that run under UltraSonix must be modified to use different versions of the MIT X Window System libraries, that understand the Remove Access Protocol (RAP). This protocol is used to communicate between applications and UltraSonix.

4.2 The Remote Access Protocol

The Remote Access Protocol (RAP) is a binary protocol that is exchanged between applications (called "clients") and external programs, such as screen readers and automated testing tools (called "agents").

The protocol contains three different types of messages:

Requests	Travel from agent to client, and must be answered with a Reply.
Reply	Travel from client to agent, in response to a Request message.
Notify	Travel from client to agent, and are unsolicited.

Requests and replies are used when the agent needs to know some particular piece of information about a client: for example, the position of a specific widget on the screen. Notifies are generated asynchronously by clients whenever their state changes (when widgets are created, for example).

The specifics of the protocol are described in Appendix A, "RAP Protocol Specification," and are also available on the World Wide Web, at the URL:

http://www.cc.gatech.edu/gvu/multimedia/x-agent/index.html

The "X-Agent" mailing list run by the X Consortium is used for discussion of RAP and RAP-releated ideas.

The current RAP implementation is available from Georgia Tech at:

ftp://multimedia.cc.gatech.edu/pub/rap-sample.tar.Z

The protocol uses ICE (Inter-Client Exchange) as its transport, and expects to work with the HooksObject present in the R6 libXt implementation, and the XESetBeforeFlush client-side extension in the R6 libX11 implementation.

4.3 Rendezvous

(NOTE: This section describes the rendezvous mechanism currently used by UltraSonix. A new rendezvous mechanism is being drafted by the X Consortium as an extension to the Inter-Client Communication Conventions Manual (ICCCM).)

When UltraSonix starts, it creates an unmapped window as a "holder" for a property. This property, called "IceNetworkIds," contains the ICE network address on which UltraSonix will listen for incoming client connections. The value of this property is a comma-separated list of ICE network IDs (see the ICE library documentation for details on how this list is created). At this point, UltraSonix begins listening on the ICE network IDs it has published, indicating that it is now willing to accept connections from clients.

Next, UltraSonix solicits SubstructureNotify events from the root window. UltraSonix detects the presence of a new application by looking for its top-level window to be mapped.

Once a new child of the root window is mapped, UltraSonix calls XmuClientWindow() to determine the "application" window (which will be a child of the window manager frame mapped directly as a child of the root window). Once this window has been identified, UltraSonix generates a ClientMessage event to the application. The format of this ClientMessage is as follows:

window	The application's window.
type	ClientMessage
message_type	Atom("ExternalAgent")

format	32
data.l[0]	Atom("RAP")
data.l[1]	Atom("IceNetworkIds")
data.l[2]	The agent's window
data.l[3]	0
data.l[4]	0

The first longword of data contains the atomized name of the protocol the agent wishes to speak (in this case, "RAP"). The second longword contains the atomized name of the property on the agent's window containing the agent's ICE listener address, and the third longword contains the window ID of the agent window on which this property exists. The remaining data areas are not used.

When the client receives this message, it should retrieve the value of the property specified on the agent's window, and open an ICE connection to the agent at the address specified there.

4.4 Client-Side Support

Clients must be extended to support RAP and RAP rendezvous. The specific infrastructure requirements for clients are:

- They must be based on Xt and X11.
- Their Xt must support the HooksObject (first present in X11R6).
- Their Xlib must support the XESetBeforeFlush client-side extension.
- They must be linked with the RAP client-side protocol library.
- They must have an event handler installed on their application shells to respond to the ClientMessage events used for rendezvous.
- (OPTIONAL) XGCList support.

UltraSonix ships with a version of the MIT X11R5 libraries that has been extended with the R6 implementation features (HooksObject and BeforeFlush extension). These libraries also have a dependency on libRAPclnt (the RAP client-side protocol library), so this library will be loaded automatically whenever libXt or libX11 is used. We have extended the ShellRealize method in Shell.c to install the ClientMessage event handler on application shells.

When the client receives the rendezvous message, it installs various callback procedures into the HooksObject callback list, and a flush procedure into XESetBeforeFlush(), along with several other routines. These routines are responsible for generating the asynchronous Notify messages to the agent. The client also retrieves the agent's window property and establishes an ICE connection to it.

At this point the client is fully connect and will respond to any RAP messages sent to it, and will generate RAP Notifies.

The optional XGCList support mentioned above is a client-side extension that we have developed to maintain a client-side mapping from GC IDs to GC structures. If this extension is enabled, RAP agents will be able to ask clients for the graphics context information corresponding to GC IDs.

4.5 UltraSonix Support

Support for the RAP protocol in UltraSonix is divided into three components: A low-level protocol library, a RAP Listener class (responsible for responding to new RAP connections), and a RAP class (responsible for dealing with single RAP clients).

4.5.1 RAP Agent Library

The RAP Agent library, libRAPagnt, is a low-level library designed to encapsulate the RAP protocol and hide it behind an easy-to-use API. RAP Agent library APIs allow callers to generate RAP messages to a client and, when messages are received, break their contents out into structures for easy processing.

UltraSonix links against this library for protocol processing.

4.5.2 RAP Listener Class

The RAPListener class is responsible for listening for incoming RAP connection requests from clients. There is one instance of RAPListener in all of UltraSonix.

Whenever UltraSonix detects a newly-mapped window, it calls the InitiateConnection() method on RAPListener. RAPListener begins the rendezvous protocol (issuing client message events, etc.) and waits on the client to respond by connecting to the listener address.

Once a client has connected, RAPListener begins the ICE protocol setup and negotiation phase. At this point, the client and the agent are only "partially" connected: a communications channel is established but they have not agreed on protocol versions, etc.

Once the negotiation process has completed, RAPListener accepts the connection and creates a new instance of the RAP class to deal with the protocol needs of this one client.

4.5.3 RAP Class

The RAP class is responsible for communicating with one particular client. Thus, there is a separate RAP instance for each client currently connected to UltraSonix.

RAP instances are created by RAPListener as new connections come in from clients. When a RAP object is first created, it ensures that it is completely connected to its client, and creates a new Client instance in the application model manager. At this point, the RAP object is responsible for all communication with this particular client.

Once connection is fully established, UltraSonix will (via the AppModelMgr) generate a series of initial messages to the client. Currently, these messages include:

GetGCValuesRequest	Download all GCs in use by the client.
HelloRequest	Confirm application toplevel window ID.
FullQueryTreeRequest	Download the application's widget hierarchy and resources.
SelectEventRequest	Tell the applications which X events we wish to be forwarded to us.
SelectRequestRequest	Tell the applications which X requests we wish to be forwarded to us.

The events UltraSonix solicits from applications include:

KeyPress	Used for echoing user input.
ButtonPress	Used for echoing user input.
EnterNotify	Optional generation of focus change feedback.
MapNotify	Ensure that the model is consistent with respect to maps/unmaps.
UnmapNotify	Ensure that the model is consistent with respect to maps/unmaps.

The requests UltraSonix solicits from applications include:

X_CreateGC	All of the GC, ImageText, PolyText, CopyArea, ClearArea, and FillRectangle requests are used by ProtoTextRep to keep its model of application text areas up-to-date.
X_ChangeGC	
X_CopyGC	
X_FreeGC	
X_ImageText16	
X_PolyText16	
X_ImageText8	
X_PolyText8	
X_ClearArea	
X_CopyArea	
X_PolyFillRectangle	
X_ReparentWindow	Ensure that the model remains consistent when windows are reparented.

The RAP object is a subclass of FDInterest, and as such has a HandleActivity() method on it for dealing with messages (either Replies or Notifies) received from the client. The HandleActivity() method on the RAP object is essentially where the "brains" about how to interpret the RAP protocol live. The code here must be able to handle and make sense of any RAP messages generated by the client.

The RAP object also has methods on it for generating requests to the client.

The RAP object flags itself for deletion (via the FDInterest mechanisms) and shuts itself down whenever the ICE connection to its client is severed.

5.0 TEXT MODELING

5.1 Introduction

One of the most important tasks of any screenreader is capturing and modeling the textual information present on the computer screen. In many regards, capturing and presenting textual information accurately is more difficult than capturing and presenting the graphical controls of a computer interface. This difficulty is due to the fact that the graphical controls in an interface typically have a one-to-one correspondance with the programming constructs used to create the interface (widgets correspond to on-screen push buttons, for instance).

In comparison, in most toolkits for graphical applications only rudimentary support is available for creating rich textual presentations. Typically the programmer is left to build the facilities for "beyond basic" text display by hand.

Thus, it is left to the screen reader to attempt to detect whether text is being rendered in two-column format, or whether wide spaces between lines represent blank lines or simple double-spacing.

5.2 Text Modeling in UltraSonix

In UltraSonix, text modeling is performed by the TextRep (for "text representation" class hierarchy. A base class, called TextRep, provides an API for retrieving textual information from the model. TextRep itself is an "abstract base class" which can never be instantiated directly. Instead, one of its subclasses is always instantiated whenever a model for text is required.

There are two TextRep subclasses provided by UltraSonix. The first is called ProtoTextRep. ProtoTextRep is a TextRep specialized for capturing information from the X protocol stream. ProtoTextReps are used as a "last chance" mechanism for capturing text being drawn to a window.

The second TextRep subclass is called XmTextRep. XmTextRep is specialized for representing text within a Motif XmText widget (which is perhaps the most common widget for text display in a Motif or CDE desktop environment). The Motif text widget provides mechanisms for retrieving text which are more robust and more powerful than simple X protocol monitoring; XmTextRep is used whenever a TextRep is needed for a Motif text widget; ProtoTextReps are used in other cases.

5.3 How UltraSonix Creates TextReps

Text may be drawn to any widget in a graphical application: push buttons, labels, multi-line text widgets, single-line text fields, and so on. UltraSonix will only create a TextRep for widgets for which the "allowTextMode" attribute is set to TRUE (this attribute may be either set for a particular widget via an object template or, more commonly, for an entire class of widgets via a class template).

Note that UltraSonix "text mode" is only available for objects which have a TextRep associated with them. The creation and maintenance of TextReps does add overhead to the system at run-time however. They should only be created for objects for which the user may have a need for rich textual interaction and screen-reader functionality.

As the UltraSonix off-screen model is being updated via RAP messages which indicate the creation of widgets in the application, UltraSonix checks the name and class of the widget to see if the user has indicated that the object should support text mode. If there is such an indication, UltraSonix creates the appropriate TextRep subclass (XmTextReps for XmText widgets, and ProtoTextReps for all other widget classes).

NOTE: In the future we may support the ability to specifically indicate via template attributes which TextRep subclass should be created for particular objects.

5.4 TextRep Basics

The TextRep classes provide a row-column oriented model of text. Text is stored in lines consisting of a number of characters. Some lines may be blank.

Segments of text within a TextRep are represented by TextData objects. TextData objects store a string of text, the attributes associated with the characters in that string, and the length of the string. Attributes are stored in a class called TextAttr. See the section "Support Classes" below for more details.

NOTE: current versions of the TextData objects do not support "wide" (16-bit) characters; only 8-bit.

5.5 The TextRep Programming Model

The TextRep classes provide only a storage model for the textual data which is *currently* present within a window on the display. TextReps provide no user-oriented output abstractions (such as filtering or "read by line" capabilities). The user interface to the data stored in a TextRep is created by a ScreenReader instance which provides functions for the tokenization, retrieval, filtering, and presentation of the text stored in a TextRep.

Note that separating the storage of text (in the TextRep class) from the presentation of the text (via the ScreenReader class) allows us to keep the interface code (and in fact, the rest of the UltraSonix system) the same, even if new TextRep classes are available in the future. New subclasses of TextRep will support the common TextRep API and will thus "fit in" with the rest of the system without modification.

The basic TextRep programming model only provides functions for text retrieval from the model. This is because the facilities for inputting text into the TextRep will vary from subclass to subclass (for instance, ProtoTextReps fill their models via protocol monitoring; XmTextReps fill their models via tracking resource updates).

The basic text retrieval functions are:

```
TextData    *GetData() const
TextData     GetLine(int linenum) const
TextData     GetSegment(int linenum, int start, int size) const
```

All three APIs return TextData objects. The first returns all of the data in a particular text object as an array of TextData instances. (The length of this array may be determined via the Rows() method on the TextRep). ,The second function returns all of the text in a particular line (the line numbering starts at zero). The third returns a segment of text beginning at the specified line number and character within that line, and extends for 'size' characters (the actual number of characters returned may be less, if you ask for more text than is available in the TextRep).

Other APIs are available to convert from the character coordinate system to pixel coordinates, and for retrieving the position of the application's cursor (when supported):

```
virtual int     GetPixelCoords(int line, int pos, int& x, int& y);
virtual void    GetAppCursor(int& row, int& col);
```

5.6 ProtoTextRep

The ProtoTextRep is a subclass of TextRep which maintains its model of text by monitoring the low-level X protocol traffic for rendering text on the screen. The following are the text-related requests in the X protocol which are monitored by ProtoTextReps:

ImageText8
ImageText16
PolyText8
PolyText16
ClearArea
CopyArea

When a ProtoTextRep is created for a particular object, it retrieves the RAP object for the client application that the object is a part of. If the new ProtoTextRep is the first ProtoTextRep in this particular client, then it calls the RAP object to solicit these text-related requests from the client application. They are not solicited by default since they add significantly to the overhead of the system; they are only requested when the first ProtoTextRep is created for a given application. Once solicited, all text-related requests generated by the application will be sent to UltraSonix--even ones for windows which do not have TextReps associated with them (push buttons for instance).

Note that requests related to GCs (Graphics Contexts) are used primarily for ProtoTextReps, but they are solicited whenever UltraSonix first connects to an application. GC-related traffic is generally low, and this information could be used in the future for tasks unrelated to ProtoTextRep modeling.

Internally, ProtoTextReps store text as an array of pointers to Line instances. Each Line maintains a TextData instance which represents the text in that line on the screen, the baseline (lower left corner) X and Y position where the text starts, and the upper Y coordinate which represents the upper edge of the bounding box around the line.

Whenever new text is drawn, the baseline Y position is determined and the array of Lines is scanned. If the baseline Y position is equal to an existing line, then the new data is considered an update of the existing line and the new data is merged in.

Otherwise, the baseline Y and upper Y positions of existing lines are checked. The ProtoTextRep does not support overlapping text, so if the new text is being drawn "over the top" of existing text, an error is reported. Otherwise, a new Line instance is created and inserted into the model.

One of the most difficult tasks of modeling text via protocol monitoring is determination of blank lines. In an effort to conserve network bandwidth, some applications will not transmit a "draw blanks" message to render nothing on the screen. This causes difficulty in determining whether a given region on the screen is a blank line, or whether lines are simply widely spaces.

For example, consider two lines of text, each with a height of 8 pixels. Suppose these lines are separated by 40 pixels of "blank space." The ProtoTextRep must decide whether these 40 pixels represent empty lines that the user would be able to navigate to (and if so how many), or whether this space represents non-navigable interline spacing.

The algorithm for blank line determination used by the ProtoTextRep is this: the ProtoTextRep maintains information about the current minimum interline spacing in use in the model. Each time a new line is inserted into the model, the minimum interline spacing in effect is recalculated. Each time the minimum interline spacing decreases, the text model is recalculated and blank lines are

inserted into the model between existing lines. The model interprets the decrease in minimum interline spacing (that is, text is being drawn between two previously existing lines) as an indication that its current supposition about interline spacing was wrong. It "renumbers" the lines based on the new information, inserting blank lines as necessary.

Note that there are a number of limitations with the algorithms used by ProtoTextRep:

- Only fixed-width fonts are supported. If font width changes were supported, reliable determination of column information in the presence of blanks would be very difficult.
- Font changes are supported as long as widths never change.
- Overlapping text is not supported.
- Only the text currently visible on the screen is reliably modeled.

The ProtoTextRep class models the location of the application cursor by tracking the X_PolyFillRectangle request to look for "block cursor" drawing in the text area.

5.7 XmTextRep

(NOTE: XmTextRep is not implemented.)

5.8 Support Classes

Several classes are used by TextReps which are exposed to users of the class. The most important of these is TextData, which encapsulates a segment of text and the attributes of that text.

Textual attributes (color, font, etc.) are represented by TextAttr objects. These are stored within TextData instances and are accessible through them.

Visual debugging of TextReps is supported via the TextRepDebug class. Associating an instance of TextRepDebug with any TextRep provides a visual indication of what the TextRep instance "believes" is currently in its model.

5.8.1 TextData

The TextData class provides a representation for a sized segment of textual data. Methods provide access to the character string stored in the instance, the attributes of those characters, and the size of the segment. Currently only 8-bit characters are supported by TextData.

Since copying and exchanging text segments is performed very often, the TextData class has been implemented to be as efficient as possible. The class is implemented as a wrapper around a pointer to a TextDataRep instance, which is hidden from users of TextDatas. TextDataRep maintains the actual data of the segment and a reference count. Copying and assigning TextDatas results in a simple pointer assignment and an update of the representation's reference count.

TextDatas implement copy-on-write semantics. Whenever the data in the class is accessed for writing, a copy of the representation is made to prevent other TextDatas which share the representation from having their data upated without their knowledge.

The basic APIs supported by TextData include the following:

```
//
// Provide non-const (writable) access to the character data
// within a TextRep. These are expensive because they cause a
// copy of the text data to be performed to ensure safety.
//
operator char*()
char *Chars();
//
// Provide const (read-only) access to the character data
// within a TextRep. These are very inexpensive.
//
operator const char*() const;
const char *Chars() const;
//
// Return the TextAtr which corresponds to the given character
// position. The first version is non-const (expensive),
// while the second is const (cheap). Both of these methods
// will raise an xmsg exception if the position is out of
// bounds.
//
TextAttr& Attr(int position);
const TextAttr& Attr(int position) const;
//
// Return the length of the segment, in characters.
//
int Length() const;
```

For convenience, TextData instances may be freely copyied, assigned, and concatenated (via operator+=). See the code for the full set of methods available on TextData objects.

The character arrays returned from TextData instances are guaranteed to be NULL-terminated.

5.8.2 TextAttr

The TextAttr class represents the attributes of a given segment of text. Currently TextAttrs are simply wrappers around the GCValue class which encapsulated X Graphics Contexts.

No special methods are currently available for determining or comparing TextAttrs. Instead, to perform comparisons, you must extract the GCValue from the TextAttr and use the standard X GC operations.

5.8.3 TextRepDebug

TextRepDebug provides a visual debugging tool for TextReps. Code which uses TextReps should never see TextRepDebug instances directly. Instead, to enable debugging of a particular TextRep, the user calls TextRep::Debug(TRUE) to set the debugging state for that instance to TRUE. Calling TextRep::Debug(FALSE) disables debugging; debug state can be determined via calling TextRep::Debug().

When debugging is set to TRUE, the TextRep creates an instance of TextRepDebug and retains a pointer to it internally. There is only one TextRepDebug class, regardless of the type of TextRep which created it; TextRep subclasses interact with TextRepDebug via a well-defined API which allows TextRepDebug to visually display TextRep contents for any class of TextRep.

When a TextRepDebug is instantiated it will create a new window on the display which will be kept "in sync" with the TextRep contents. No user intervention is required.

6.0 THE SCREENREADER

6.1 Introduction

The ScreenReader object provides tokenization functionality for the textual data whose onscreen representation is stored by the TextRep object. Each ScreenReader object (one may exist for each area of text on the screen) contains a pointer to a TextRep object, which provides an interface to the raw textual data displayed onscreen at a given snapshot of time.

Each ScreenReader maintains two cursor abstractions (this may be possibly expandable in the future) as CursorRep objects. These objects are coupled to the TextRep object stored in the ScreenReader object. The CursorRep object provides a notion of row and column access to the raw data obtained from the TextRep, as well as absolute and relative positioning, and bounds-checking and error-raising via exceptions. Each cursor may be queried and manipulated independently, however, access to the cursors from outside the ScreenReader object is controlled by the variable operation_mode, which can be set The CursorRep object serves as the interface between the ScreenReader and TextRep objects. Using the CursorRep objects, the ScreenReader may perform query operations on the text relative to the current cursor positions. Query operations include next/previous word, line, etc.

The ScreenReader also contains "filter(s)", which modify the presentation of text parsed by the ScreenReader object by either altering the information presented directly, or by altering the manner of presentation on a particular output device or devices.

6.2 Functional Overview

The ScreenReader object provides the following functions:

- Read by character, word, line, sentence, or paragraph
- Move by character, word, line, sentence, or paragraph
- Synchronize cursors
- Perform text processing

Each function is briefly described below:

Move: the appropriate lexical unit (character, word, line, sentence, or paragraph) is read from the current cursor position. The cursor is updated to point to an appropriate location following the read. For example, reading a sentence would cause the cursor to be updated to the end of the sentence after it has been read.

Read: similar to Move, as above, except that the cursor position is not altered.

Synchronize cursors: set the location of one cursor to be the same as the other.

Perform text processing: each ScreenReader object has an associated filter object which performs default processing on the text item returned before it is returned to the output device.

6.3 Implementation Overview

6.3.1 CursorRep

The CursorRep object is serves as the interface between the TextRep and ScreenReader objects. It provides a row/column interface to the text block represented in the TextRep, as well as increment/decrement operations to scan forward and backward, treating the text block as a stream. The text data in the TextRep associated with the cursor can be accessed from the current position (sequential access), or randomly by specifying a row/column position (random access). The CursorRep also contains a copy of the text data for the current line of data.

6.3.1.1 Terminology

Text block: The raw data stream presented by the TextRep object. Each line of screen output is concatenated into a single string, with no delimiter characters between lines (lines are therefore referenced positionally, based on a calculated offset from the start of the text area, rather than semantically).

Text offset: Since each line of displayed text is assumed to be of constant length, the offset from the start of the text block for a given character may be calculated using the formula (row x line_length) + column. An offset as well as a row/column representation of the cursor position is stored in the CursorRep object.

Text bound: The upper bound on the offset representation value for the CursorRep with its current snapshot of the TextRep object, or (line_length * number_of_lines)-1. The lower bound for the offset representation is always 0. The bound values are inclusive (i.e. they themselves are legal, but values below 0 or above the upper text bound are not). Attempts to access positions outside the text bound will result in a thrown exception.

6.3.1.2 Overview

All of the functionality of the ScreenReader object is implemented via the CursorRep object. The CursorRep acts as a scanning head which may advance forward or backward in the text block. The delimiter string (see below) determines how the ScreenReader determines where word, sentence, and paragraph boundaries occur while it reads data from the TextRep via the CursorRep.

Typically, ScreenReader functions will be invoked by keystrokes by the user, which are bound to TCL functions defined in text.tcl. These functions then invoke the sreader() function in Interp.cc, which calls methods of the ScreenReader class.

6.3.2 ScreenReader

6.3.2.1 Terminology

Lexical unit: The lexical units which are parsed by the ScreenReader object include characters, words, lines, sentences and paragraphs.

Cursor: Each ScreenReader object contains two cursors -an edit cursor and a review cursor. The various navigation and fetch functions work with either cursor.

Delimiter string: Characters in the delimiter string define the set of characters which serve as delimiters for the text returned by the TextRep object. Due to implementation, characters which are used frequently as delimiter characters should be placed at the front of this string to improve performance.

Forward/backward: The forward direction corresponds to the direction in which text is normally read, that is, left to right across the screen. Thus, functions which retrieve the "next" lexical unit read "forward", while functions to retrieve the "previous" lexical unit read "backward".

6.3.2.2 Overview

The ScreenReader allows the user to navigate via the cursors and fetch current, next, and previous lexical units from the current cursor position. The methods which provide this functionality are called in a hierarchical fashion. That is, the function to parse characters and words is based on the CursorRep object, the function to parse sentence-level constructs is based on the function to parse words, and the function to parse paragraphs is based upon the function to parse sentences. The tokenization algorithm for each lexical unit is described below:

Word: Depending on the desired direction of the scan (current lexical unit requested is treated as a forward scan), the cursor moves backward to the start of the current word or forward to the end of the current word. The scan then proceeds to move forward to the end of the word or backward to the start of the word, terminating the search when the text bound or a delimiting character is reached. The cursor is updated to point to the final character in the word for a forward scan, the first character in the word for a backward scan.

Sentence: Words are read backwards from the current cursor position until a word ending in a terminal character is found; this word is discarded as being part of the previous sentence to the current one. Next, words are read forward from the current cursor position until a word ending in a terminal character is found, however, this word is retained, since it is part of the current sentence. The words read by this routine are stored by the procedure until the complete sentence is found, and the results of the individual searches are concatenated together. The cursor is updated to point to a character in the last word in the sentence for a forward scan, the first word in the sentence for a backward scan.

Paragraph: Sentences are read backwards from the current cursor position until the word immediately preceding the first word of the current sentence differs in its row position from the current sentence by at least two lines -this indicates the presence of a blank line between the previous sentence and this one. Similarly, sentences are read forward from the current cursor position until the next word past the end of the sentence differs by at least two lines. The sentences read by this routine are stored by the procedure until the complete paragraph is found, and the results of the individual searches are concatenated together. The cursor is updated to point to a character in the last word of the last sentence of the paragraph for a forward scan, the first word of the first sentence of the paragraph for a backward scan.

Data returned to the ScreenReader is encapsulated in the CursorRepData structure, which appears as follows:

```
class CursorRepData {
  int startx, starty, endx, endy;
  TextData *tdata;
}
```

The start and end parameters are intended to store the row/column position of the start and end positions for the data stored in this block (currently, this is unimplemented). The TextData structure is defined in TextRep.cc and contains both character information as well as the text attribute information for the data returned from the TextRep.

6.3.3 Filters

Filters are read via the parsing and lexing routines in filter.y and filter.l, which are converted into the files filterparser.c and filterlexer.c using sed. These routines initialize the data structures which are used to store the actions to be performed by the filters.

A filter consists of a TextFilter object, which is defined as follows:

```
class TextFilter {
  [...]
 char *name;
 int active;
 FilterEntryPtr *commands;
 int num_commands, max_commands;
}

class FilterEntry {
  [...]
 mRegexp r;
 int action, action_arg;
 char *arg;
 } *FilterEntryPtr;
```

Hence, each TextFilter is a named structure of num_commands FilterEntrys. If it is active, it will be applied to data returned by the ScreenReader which uses the filter with this name.

Each FilterEntry describes a single action to be performed by the filter. The mRegexp member describes a regular expression which will be pattern matched against the data passed into the filter. The action, action_arg, and arg fields describe possible actions to take when this regular xpression matches successfully. Currently, only the arg field is significant, and represents the character string which should be used to substitute into the data string when a pattern match occurs. However, the action code may be used to define a set of actions which may occur instead (.i.e. switch(action) { }), using the action_arg and character as arguments to whatever functions are desired. For example, the current parser for the filter contains rules for operations, such as LANGUAGE, PITCH, RATE, VOLUME, or VOICE (these operations are unimplemented). A separate action code may be defined for each operation, which can then be coded in TextFilter::Filter.

7.0 INTERPRETED RULES

8.0 CONFIGURATION SUBSYSTEM

UltraSonix uses a fairly complex configuration subsystem to allow users and administrators to change the behavior of the system. There are two primary components of this system: the configuration mechanisms and the template mechanisms. The configuration mechanisms are used

to control overall system parameters; the template mechanisms allow users to specify behavior on a per-widget, per-class, or per-application basis.

This section describes the internals of the configuration and template subsystems.

8.1 The Template Files

Template files are text files that contain attribute information for different objects.

Template files are usually denoted by the .tmpl extension in their name. A template file contains one or more template definitions, although it is customary to only have one definition per file and to have the filename correspond to the definition name.

The following is an example of a complete template file with one definition:
(from Athena/Command.tmpl)

```
classTemplate Command {
  navigable         = TRUE
  shell             = FALSE
  sound             = "/net/hc22/selbie/dink.au"
  unsafeResources   = { "sensitive", "mappedWhenManaged" }
  volume            = 90
  muffle            = [PositionMuffle]
  sensitive         = [SensitiveProc]
  mappedWhenManaged = [MapWhenMgdProc]
  speakOnEnter      = [AthenaLabelSpeaker]
  speakOnInfo       = [AthenaInfoSpeaker]
}
```

The word "classTemplate" identifies the type of template that is being defined. "Command" is the class of the object whose attributes are listed. Attribute names are listed in the left hand column between the braces, and attribute values can be one of 4 data types - boolean, integer, string, or stringlist.

In this example, the navigable and shell attributes are both boolean attributes, thus their values are defined on the right hand side of the assignment statement to be either TRUE or FALSE.

Sound is an attribute of type string, therefore it's value is enclosed withing double quotes ("") to identify it as such.

Volume is an attribute of type integer. It's value can be any number between -(2^31) to +(2^31-1).

UnsafeResources is of type string list, and is enclosed in braces.

The other attributes listed below are all defined by a name enclosed within brackets ([]). These attributes all have valid data types, but are defined by a TCL procedure. (The TCL procedure is the name in brackets). Whenenver the value of one of these attributes is required, the named TCL procedure will be evaluated at run time to return the result.

The above attributes are all members of a standard set of attributes that can be defined in any class or object template definition.

There are three different types of template definitions. "objectTemplate" definitions refer to a specific object in a specific program. For example, if a template is defines a "objectTemplate XMailTool.Command {..." it refers to a particular Command Button widget in the XMailTool application.

If the definition is listed as "classTemplate Command {..." then the attributes of the definition refer to any Command Buttons that are not superseded by an object tempalte definition.

Both class and object templates support the same set of attributes; they merely provide two different mechanisms for associating those attributes with objects.

A third type of template definition is that of an "appTemplate," which allows the configuration of per-application attributes. Application templates support a different set of attributes than class/object templates.

When UltraSonix starts, it retrieves the list of directories containing template files from its configuration file, and loads all files ending in the ".tmpl" extension in those directories. These files are parsed into an internal (in-memory) format, which is stored as a dictionary of objects of class Template.

8.2 Defaults and General Configuration File

UltraSonix may be run on different machines with different configurations. Thus, a file exists for the user to specify default variables for different parts of the system. The file, "mercator.config" is used for assignment of various variables as well as to contain fallback values for undefined template attributes.

If at any time, UltraSonix needs the attribute value for a particular object, it performs the attribute look up in the following order:

1) If a template exists for the object's specific name, and the attribute is defined, then return the value (calling a TCL procedure, if the attribute has been specified that way).

2) If an object template does not exist, or if the attribute was not defined in the object template, then UltraSonix checks to see if a template exists for the class that the object belongs to. If a class template exists, and the attribute is defined, then return the value. (Again, calling the TCL procedure, if necessary).

3) As the final fallback, UltraSonix will look in the file "mercator.config" for attribute values not defined in a Template file. Default attribute assigments in this file may also be TCL procedures. If a value for a standard attribute is not given here, UltraSonix will not start--the user *must* provide fallback values for all template attributes in use.

The following is a sample portion of the mercator.config file. The format of the file is a series of assignment statements with the variable name on the left, followed by an equals sign, followed by the assignment value: (very similar to the lines of a template file)

```
###
### Default values for class templates
###
defaultClassNavigable        = TRUE
defaultClassShell            = TRUE
defaultClassAllowTextMode    = FALSE
defaultClassSound            = "/net/hc22/selbie/ezek2517.au"
defaultClassSpeakOnEnter     = ""
defaultClassSpeakOnInfo      = ""
defaultClassLooped           = [fooProcedure]
```

8.3 Declaring New Attributes with mercator.attrib

A file exists that allows the user to "declare" new attribute variables by specifying an identifier name for the template files, an identifier name for the mercator.config file, its data type, and the type of templates the attribute will be valid on. This ability is very handy advance customization--the ability to add new template features--without having to recompile UltraSonix.

The format of the "mercator.attrib" file is a series of lines with four fields. Each line represents one declaration. The following is a sample portion of the file that declares some of the "standard" attributes.

```
# UltraSonix Attribute definition file
#
# NAME        TYPE       CONFIG FILE NAME          TEMPLATE TYPE
# ###########################################################################
navigable     BOOLEAN    defaultClassNavigable     CLASS
volume        INTEGER    defaultClassVolume        CLASS
speakOnEnter  STRING     defaultClassSpeakOnEnter  CLASS
foo           INTEGER    defaultAppFoo             APP
bar           STRINGLIST defaultClassBar           CLASS
```

The first identifier on each line is the name of the attribute identifier as it will appear in the template definition files.

The second field is the data type of this identifier. Valid types are BOOLEAN, INTEGER, STRING, or STRINGLIST.

The third field represents the name of the identifier in the mercator.config file. The differentiation between the template attribute name and the config file name allows the user to have separate default values for both class/object and application attributes. Notice that the identifier "foo" is defined to be used in both Class/Object and Application templates, but has separate identifiers in mercator.config.

The fourth field may either be CLASS or APP to indicate what type of template the declared attribute is to be used with. CLASS indicates that it will be use in either object or class templates

(object and class templates always have the same attributes). APP indicates that it will be used in Application templates.

Thus to declare a new attribute, an entry in "mercator.attrib" must be added as well as a corresponding entry in "mercator.config". Undefined (ala bad) events may occur is an attribute does not have a fallback definition in or procedure in mercator.config. (disallow this in later work)

After these entries are added, UltraSonix needs to be restarted for these changes to take effect.

To be added: Facility to resource all template and configuration files during run time. Thus, a restart for changes to take effect won't be necessary.

8.4 Adding New UltraSonix Configuration Variables

If at any time a programmer maintaining UltraSonix needs to create a new user-definable varialble that will be referenced throughout the system, he or she only needs to add an entry in the Configuration (mercator.config) file.

For example, suppose a driver for a new Braille terminal is written. The driver is a shared object file called "MyBraille.so". The shared object may need other parameters passed to it when it is loaded: the name of a braille server to start, or some additional configuration variables for example.

UltraSonix will parse and store *any* entries in the configuration file that it encounters. So no additional programming is needed to support parsing of new configuration attributes; simply add the new attributes to the configuration file:

```
myBrailleServer    = "/my/home/dir/server"
myBrailleColumns  = 80
```

8.5 Accessing Attributes Programmatically

Developers have both C++ and TCL interfaces to retrieving attributes from the configuration and template subsystems. The example below shows how to retrieve data from the configuration file for the "MyBraille" examples above.

Inside MyBraille.so, the programmer must include "Config.h" to access this new attribute.

The easiest way to access the attribute is via the "direct" APIs which assume the type of the attribute:

```
char *brailleServer = Config::GetString("myBrailleServer");
int brailleColumns = Config::GetInteger("myBrailleColumns");
```

(There are similar APIs for retrieving boolean and string list data.)

These routines have undefined return values if the attribute does not exist or if its type does not match the "expected" type. A safer way to fetch the data is to return it into a FieldData instance, like this:

```
FieldData *data = Config::GetDefaultByName("myBrailleServer");
char *brailleServer;
```

```
if (data && (data->Tag == STRING))
    brailleServer = data->String();
else
    // ...do some error reporting...
```

FieldData is a special class that is used to represent the values of configuruation or template attributes; it may represent data in any of the supported types. The GetDefaultByName() API will return a NULL FieldData pointer if the attribute does not exist. After the pointer has been returned, callers can determine the type of the value as it exists in the configuration file.

FieldData instances are used by both the Config and the Template subsystems, and thus the APIs used to access attributes are the same for both.

8.6 Writing Templates to Files

An interpreter command and function are in place to allow for the possibiblity of interactive customization.

The interpreter command is called "writeout" and it is binded to the function "WriteOut" which is defined in Interp.cc. It can be called from TCL as well.

"writeout" has the following syntax:

```
writeout object-name filename [dictionary]
```

object-name is the name of the Template object (e.g. XmLabelGadet)

filename is the fully-qualified file name of where the output is to be placed

dictionary is an optional parameter to specify which Template dictionary to search for attributes for. By default all dictionaries are searched until a matching object-name is found. They are searched in the following order: Object-Templates, Class-Templates, and App-Templates.

Caveats:

- "WriteOut" is a friend function of the Template Class. This is to facilitate access to the private dictionaries. Most of this code
- should be made as a method of the Template Class.

8.7 Re-Sourcing of Template and Configuration files

Four commands have been added to the Interpreter code (Interp.cc) to support reloading of configuration and template information.

"loadattributes" - reloads mercator.attrib
"loadconfig" - reloads mercator.config
"loadtemplates" - reloads all the template files

"loadall" - calls the above three commands in the above order

The corresponding function names that these commands bind have the same name.

9.0 DEVICE-SPECIFIC CODE

9.1 Motivation

UltraSonix "hides" the details of low-level input/output processing to make the system as portable and flexible as possible. All device-specific I/O code is externalized out of the "core" part of the system. I/O code is dynamically loaded at runtime as it is needed.

To ensure that "foreign" code will operate as expected with UltraSonix, the system requires that all loadable code conform to one of a set of "base class" APIs that define how different I/O devices will appear to the rest of the system.

By writing loadable I/O code that conforms to one of the pre-defined base classes, arbitrary device-specific code can be loaded into UltraSonix at runtime and can interoperate with the rest of the software, without the need for any changes. All existing C++ and TCL code can use the new device-specific code without even being aware of what devices it is interacting with.

This section details the loadable I/O strategy used by UltraSonix, and describes how to create new device-specific code.

9.2 Loadable Base Classes

As mentioned, all device-specific code must conform to the API specified by one of the "loadable base classes." The loadable base classes are C++ classes the enforce particular APIs for different categories of devices, currently non-speech audio, speech synthesis, and braille.

When writing new device support, developers should chose the existing loadable base class that most closely approximates the new device they are adding support for. They must provide device-specific implementations for all of the functions in the particular base class they are deriving their code from.

The following sections detail the APIs provided by the various loadable base classes.

9.2.1 The Audio Generic API

The generic audio API is defined in Audio.h. Note that the API is subject to change as new needs are identified.

The Audio superclass defines the type Audio::SoundID which can be used to refer to playing or loaded sounds. Two methods should be implemented by derived classes:

```
SoundID PlaySound(const char *sound_file, float *rate, int volume,
                  int muffle, int looped);
```

PlaySound plays the specified sound file with the given parameters. It returns a unique ID for the playing sound.

```
int StopSound(SoundID sound_id);
```

StopSound halts the play of the specified sound ID.

int StopAllSounds();

Halts all sounds currently being played.

9.2.2 The Speech Generic API

The generic speech API is defined in Speech.h. Note that the API is subject to change as new needs are identified.

The following methods should be implemented by derived classes. All should return 0 on failure and 1 on success.

int Speak(const char *text);

Speak the specified string of text.

int SpeakUnix(const char *text);

Speak the specified string of text, performing some common UNIX translations (for pathnames, and so on).

int Notify(const char *text);

Reserved for future use.

int SetSpeechRate(int new_rate);

Change the speech rate to the specified value.

int StopSpeaking();

Halt the speech synthesizer.

int SetVoice(const char *voice_code);

Change to a voice specified the provided voice code string (this will probably be implementation dependent). The voice name must be one of the supported voices for this particular hardware.

int SetLanguage(const char *lang);

Change the current language. The new language must be one of the ones supported by the hardware.

int SetGain(int newGain);

Change the output volume of the speech synthesizer.

int SetDictionary(const char *word, const char *pronunciation);

Install a synthesizer-specific pronuncation in the dictionary.

int UnSetDictionary(const char *word, const char *pronunciation);

Remove a synthesizer-specific pronuncation from the dictionary.

Capabilities& GetCapabilities();

Return an object containing the capabilities of this particular speech synthesizer: languages and voices that are supported, minimum and maximum rate, and whether software gain control is supported.

9.2.3 The Braille Generic API

The Braille generic API is defined in Braille.h. Note that the Braille base class is derived from FDInterest, as many braille terminals are also input devices. Thus, any subclasses of Braille must implement the proper FDInterest protocols, as described in the section on Handling Input.

void DisplayRaw(const char *text, int rawPosition = 0);

Display the specified text on the braille device, at the specified position. The DisplayRaw() routine treats all status and "normal" display cells the same: text will the cross status/normal cell boundary; position is relative to the first physical cell on the display.

void DisplayBraille(const char *text, int position = 0);

Display the specified text in the "normal" display area of the braille terminal, at the specified position. If the text exceeds the number of display cells, it will be clipped.

void DisplayStatus(const char *text);

Display the specified text in the status area of the braille display. If the text exceeds the number of status cells, it will be clipped.

int JumpScroll();

Return the current jump scroll enable status (TRUE or FALSE).

void JumpScroll(int jumpScroll);

Set the jump scroll status (TRUE or FALSE)l

void ToggleSumpScroll();

Invert the current jump scroll status.

void SetText(const char *text[], int numRows);

Download an array of text lines into the braille display. The

loadable object should implement scrolling through the downloaded lines.

void Translation(const unsigned char *translation);

Set the translations (byte conversions) that will be used by the loadable object.

const unsigned char *Translation();

Return the translation map currently in use.

Capabilities& GetCapabilities();

Return an object containing the capabilities of this particular braille harware: number of display and status cells, presence or absence of particular keys, etc.

9.3 Writing New Loadables

This guide provides a quick overview of writing loadable I/O modules for use with the UltraSonix software. It assumes some familiarity with C++ and screen reader terminology.

9.3.1 Basic Concepts

The fundamental goal with the loadable module support in UltraSonix is to provide a mechanism to separate device-specific code from the core of the screen reader system. Device specific code can be compiled into "dynamically loadable" modules (also called "shared object files") which are then loaded into UltraSonix at run-time. Run-time or "dynamic loading" makes it possible for UltraSonix to use I/O devices which were not available at the time the system was written.

The internals of UltraSonix are written to use "abstract" interfaces to the various I/O devices which may be present on a user's workstation. For example, UltraSonix is written to use a generic Audio object and a generic Speech object. These generic objects enforce a particular interface for that UltraSonix relies on. The functions, or "methods," provided by these objects specify the interface between UltraSonix and the device.

These generic objects do not provide any functionality by themselves. Instead, they simply provide a programming interface which can be used by the rest of the UltraSonix system.

At runtime, subclasses of these generic classes that have been compiled as dynamically loadable modules are loaded into the system. Each of these subclasses provides the same APIs as their generic superclasses, but they also provide an actual implementation for interacting with a particular device.

As an example, there may exist several subclasses of the Speech generic class implemented as dynamic modules: Dectalk, TrueVoice, and so on. When UltraSonix is run, one of these modules (as determined by the configuration file) will be dynamically loaded into the system and used in the place of the generic Speech object. Since the subclass has the same API as its generic superclass, everything works correctly.

9.3.2 Writing a Loadable Module

The first thing you must do when writing a loadable module is decide which generic superclass you wish to base your module on. You must use one of the superclasses that are already known to UltraSonix. Currently there are only two: Speech and Audio. The core of UltraSonix has been written to use these modules. Introducing new generic superclasses involves restructuring the UltraSonix I/O system so that it can decide when to use the new module.

If you wish your loadable module to "act like" an audio device, you must derive from the Audio generic superclass. Likewise for the Speech superclass. Your module must implement the methods defined by these superclasses for it to work properly with UltraSonix. See the secions below on the particulars of the generic APIs for the Audio and Speech superclasses.

Once you've decided which superclass to use, you create a C++ class derived from that superclass. In the example, suppose we have a superclass called Super declared as follows:

```
class Super {
public:
   Super();              // constructor
   virtual ~Super();     // destructor
         //
         // Subclasses must implement DoSomething to behave
         // like a Super.
         //
   virtual int    DoSomething();
};
```

To create your loadable subclass (we'll call it Sub), create a file Sub.h that contains the declaration of Sub, and a file Sub.cc that contains the implementation of Sub.

In Sub.h, provide (at least) the following:

```
#include "Super.h"
#include "Loadable.h"

class Sub : public Super {
public:
   Sub();
   ~Sub();
   int DoSomething();
};
```

Note the #include of the file Loadable.h. This file contains the definition of a macro LOADABLE_CLASS_DEFN that performs some of the "boilerplate" required work that loadable modules must implement.

Also note that you must provide a "default" (parameterless) constructor for the class.

Now in Sub.cc, you provide implementations of the methods declared in the .h file:

```
#include "Sub.h"

LOADABLE_CLASS_DEFN(Sub);
```

```
Sub::Sub()      { /* ...constructor code goes here ... */ }
Sub::~Sub()     { /* ...destructor code goes here ... */ }

int
Sub::DoSomething()   { /* ... DoSomething impl ... */ }
```

Note the call to LOADABLE_CLASS_DEFN. This is a macro which is provided in the Loadable.h header file. You MUST place this macro in the .cc file for your class to function properly as a loadable module in UltraSonix. The argument to the macro is simply the name of the class you are creating.

This macro simply creates a function called NewSub which calls the default (parameterless) constructor on the Sub class. UltraSonix relies on this function to bootstrap the loading process.

To summarize the requirements for writing a loadable module:

- Choose an already-existing superclass to base your work on.
- Include the Loadables.h header file.
- Make sure you provide a default constructor.
- Make sure you provide implementations for all of the methods required by the superclass.
- Make sure you call the LOADABLE_CLASS_DEFN macro to automatically generate the boilerplate code needed by UltraSonix.

9.3.3 Compiling a Loadable Module

The details for how to compile a loadable module will vary from platform to platform and from compiler to compiler. The guidelines below are for the Sun SPARCcompiler C++, version 4.0.1. There are some general rules which will apply to any compiler however.

First, compile the .cc files into .o files. You must specify flags to generate position independent code (PIC). On the Sun compilers, the flags to use are -G and -K pic:

```
CC -G -K pic -c Sub.cc
```

This will create the file Sub.o. Next, you must compile the .o files into a shared object (.so) file. Note that if your loadable module depends on any other shared libraries you MUST pass these on the link line via -l. If they are present on the link line, then the Solaris runtime loader will know to load them at the same time it loads your module. Without the libraries you will get undefined symbols at runtime. Also note that if any of these libraries are shared libraries, and they reside in a place other than /usr/lib, you will need to provide a "library run path" via the -R option. This path specifies directories in which the runtime linker will search for the libraries you need.

Link as follows:

```
CC -G -K pic -o Sub.so Sub.o -L/your/dir -R/your/dir -lyourlib
```

UltraSonix follows the convention that the filename of the resultant .so module MUST be the same as the class defined in that file (and the same as the name passed to LOADABLE_CLASS_DEFN). Since only the filename of the shared object module is specified in the configuration file, UltraSonix uses the filename to derive the name of the class defined in that file.

9.3.4 Configuration

Certain loadable modules may require the specification of user-customizable parameters to operate effectively. For example, the Dectalk loadable module requires information about how to start the dectalkd speech server.

We encourage writers of loadable modules to use the standard configuration file subsystem to specify user-supplied parameters. Any loadable code can retrieve attributes from the configuration file via the Config interfaces defined in Config.h. By keeping all customization information in one configuration file (rather than a multitude of device-specific configuration files), maintenance and administration should be made easier.

9.4 Existing Loadable Modules

This section describes the loadable I/O modules currently shipped with UltraSonix: how to use these modules, and how they are implemented.

The currently available modules are:

Dectalk	DECtalk DTC01 Speech Synthesizer
DectalkX	DECtalk Express Speech Synthesizer
TrueTalk	Entropic TrueTalk Software-Based Speech Synthesizer
NetAudio	Georgia Tech NetAudio audio server protocol
AudioFile	Digital AudioFile audio server protocol
Alva	HumanWare ALVA 3-20 and 3-80 Braille terminals
Genovations	Genovations external keypad

9.4.1 Dectalk

9.4.1.1 Using the Dectalk Speech Synthesizer with UltraSonix

UltraSonix supports the Dectalk speech synthesizer. To use Dectalk with UltraSonix, connect the speech synthesizer as specified by the user's manual and start the Dectalk host daemon, dectalkd, on the same host, which is specified by the environment variable DECTALKHOST. After these steps change the line starting with "speechLoadable" in the configuration file to:

```
speechLoadable                  = "Dectalk"
```

This will tell UltraSonix to look for the loadable object named Dectalk.so in one of the directories specified by the loadableSearchPath variable in the configuration file.

When UltraSonix is executed it will load Dectalk.so automatically and the user should be able to hear speech output from the Dectalk.

9.4.1.2 Check List

If there is no speech output when UltraSonix starts up, make sure the following have been carried out:

a. Connect the Dectalk to the host which will be running the dectalk daemon.

b. Start the daemon dectalkd on the above host.

c. On the user's host, set the environment variable DECTALKHOST to the name of the machine used in a. and b. (This step is not necessary if the Dectalk is connected to the same machine UltraSonix will run on.)

d. Set the speechLoadable variable in UltraSonix's config file to "Dectalk".

e. Set the loadableSearchPath variable in UltraSonix's config file to contain the directory where the Dectalk loadable object, Dectalk.so is stored.

9.4.1.3 Implementation Details

Dectalk.so is implemented as a subclass of the generic Speech object. It contains the set of api's which is specified by the Speech object's public virtual functions. The constructor Dectalk::Dectalk takes the name of the host as its parameter and opens a socket connection to the daemon running on that host. The SetVoice function takes the following one character voice parameters:

p = standard male voice
b = standard female voice
h = deep male voice
f = older male voice
k = child's voice
r = deep female voice
u = light female voice
d = whispery male voice
w = whispery female voice

The argument to the SetSpeechRate function is the number of words spoken per minute on the average. The Speak function takes a string of text to be spoken. There may be additional functionalities which are specific to the speech synthesizer used.

9.4.1.4 Server Options

The server, dectalkd, is responsible for controlling the physical Dectalk hardware. The server can be used with either the older Dectalk devices, or the newer Dectalk Express.

The server understands the following options:

-d	Enable socket debugging options.
-f	Run in foreground (no daemon mode).
-p port	Use the specified IP port number.
-t tty	Use the specified tty device.

By default, the server uses port 1330 and device /dev/ttyb. When started from UltraSonix, the server should be run with the -f option so that it can be reliably shut down when UltraSonix exits.

9.4.2 DectalkX

9.4.2.1 Using the Dectalk Express Speech Synthesizer with UltraSonix

UltraSonix supports the Dectalk Express speech synthesizer. To use Dectalk Express with UltraSonix, connect the speech synthesizer as specified by the user's manual and start the dectalk host daemon, dectalkd, on the same host as specified by the environment variable DECTALKHOST. After these steps change the line starting with "speechLoadable" in the configuration file to:

speechLoadable = "DectalkX"

This will tell UltraSonix to look for the loadable object named DectalkX.so in one of the directories specified by the loadableSearchPath variable in the configuration file.

When UltraSonix is executed it will load DectalkX.so automatically and the user should be able to hear speech output from the Dectalk Express.

9.4.2.2 Check List

If there is no speech output when UltraSonix starts up, make sure the following have been carried out:

a. Connect the Dectalk Express to the host which will be running the dectalk daemon.

b. Start the daemon dectalkd on the above host.

c. On the user's host, set the environment variable DECTALKHOST to the name of the machine used in a. and b. (This step is not necessary if the Dectalk is connected to the same machine UltraSonix will run on.)

d. Set the speechLoadable variable in UltraSonix's config file to "DectalkX".

e. Set the loadableSearchPath variable in UltraSonix's config file to contain the directory where the Dectalk Express loadable object, DectalkX.so is stored.

9.4.2.3 Implementation Details

DectalkX.so is very similar to Dectalk.so. The only modification we made was in the control characters used to implement the SetVoice, SetSpeechRate, and StopSpeaking functions. (It should be noted that the StopSpeaking function uses an undocumented Dectalk Express command: it simply sends a control-c character (ascii 3) to the Dectalk Express. This is equivalent to the [:flush all] command described in the manual. We used the undocumented command because the [:flush all] command did not seem to work on our systems.)

9.4.2.4 Server Options

The server, dectalkd, is responsible for controlling the physical DectalkX hardware. The server can be used with either the older Dectalk devices, or the newer Dectalk Express.

The server understands the following options:

-d	Enable socket debugging options.
-f	Run in foreground (no daemon mode).
-p port	Use the specified IP port number.
-t tty	Use the specified tty device.

By default, the server uses port 1330 and device /dev/ttyb. When started from UltraSonix, the server should be run with the -f option so that it can be reliably shut down when UltraSonix exits.

9.4.3 TrueTalk

9.4.3.1 Using the Entropic TrueTalk Speech Synthesizer with UltraSonix

UltraSonix supports Entropic's software speech synthesizer, TrueTalk. To use TrueTalk with UltraSonix, the license server and TrueTalk server must be set up as described in the TrueTalk User's Manual. After these steps are done change the line starting with speechLoadable in the configuration file to:

speechLoadable = "Truetalk"

This will tell UltraSonix to look for the loadable object named Truetalk.so in one of the directories specified by the loadableSearchPath variable in the configuration file.

If the TrueTalk license server and the TrueTalk speech server are up and running, and the Truetalk loadable object is under one of the directories specified by the loadableSearchPath, UltraSonix will load Truetalk.so automatically when it is run and the user should be able to hear speech output from UltraSonix.

9.4.3.2 Check List

If there is no speech output when UltraSonix starts up, make sure the following have been carried out:

a. The TrueTalk license server, elmd, is running on a host named by the user's ELM_HOST environment variable.

b. Set the TT_BASE environment variable to the TrueTalk directory on the user's system.

c. Run the TrueTalk server.

d. Set the speechLoadable variable in UltraSonix's config file to "Truetalk".

e. Set the loadableSearchPath variable in UltraSonix's config file to contain the directory where the TrueTalk loadable object, Truetalk.so is stored.

f. Make sure the audio device is not busy.

9.4.3.3 Implementation Details

Truetalk.so is implemented in the same fashion as Dectalk.so, as a subclass of the generic Speech object. Truetalk.so contains the set of api's which conform to ones specified by the Speech object's public virtual functions. Currently the SetVoice and the SetSpeechRate functions follow Dectalk's convention, where the voices are specified by a single character and the speech rate is specified in words per minute. The voice specification currently uses these values:

p = standard male voice
b = standard female voice
h = deep male voice
f = older male voice
k = child's voice
r = deep female voice
u = light female voice
d = whispery male voice
w = whispery female voice

9.4.3.4 Compatibility Issues

Since TrueTalk is a software speech synthesizer, it will require an audio device on the user's machine. This means that if TrueTalk is used with an audio server such as netaudio, which does not mix more than one audio inputs the user would need to have two audio devices, which is currently the case.

9.4.4 NetAudio

9.4.4.1 Using the NetAudio System with UltraSonix

The NetAudio.so loadable module provides support for the Georgia Tech NetAudio-2 non-speech audio server. This server was developed at Georgia Tech specifically to support UltraSonix. Note that it bears no releation to the NetAudio server from NCD, despite the name.

NetAudio-2 supports various filtering and signal processing operations within the server code; it only runs on Sun SPARCstations.

For more information on NetAudio-2, see the WWW page at:
http://www.cc.gatech.edu/gvu/multimedia/NetAudio.html

To enable support for NetAudio-2, set the following attribute in the mercator.config file:

```
audioLoadable = "NetAudio"
```

The NetAudio.so loadable module will (optionally) start the NetAudio-2 server on the local machine.

9.4.4.2 Implementation Details

When loaded, NetAudio.so will optionally attempt to start the NetAudio-2 server (netaudiod) on the local machine. If the environment variable NETAUDIOHOST is set, NetAudio.so will attempt to connect to netaudiod on the specified machine. If NETAUDIOHOST is not set, NetAudio.so

will retrieve the value of netaudioServer from mercator.config, and execute this command, assuming that it will start a new netaudiod.

NetAudio.so also retrieves the value of netaudioTimeout from mercator.config. This value specifies, in seconds, how long NetAudio.so will attempt to connect to netaudiod before it gives up.

The methods in the generic audio API, PlaySound, StopSound, and StopAllSounds, map directly into the netaudiod RPC protocol.

9.4.5 AudioFile

9.4.5.1 Using AudioFile with UltraSonix

The AudioFile system is a non-speech audio server from Digital Equipment Corporation. It is freely available on the net.

Enable AudioFile support with the following line in mercator.config:

```
audioServer    = "AudioFile"
```

The AudioFile.so loadable module provides an interface to the audiofile server process that conforms to the generic audio API. Note that the currently implementation of AudioFile.so does not start an audiofile server: the server must be started "by hand" before UltraSonix is run.

9.4.5.2 Implementation Details

The audiofile server does not support the filtering and signal processing applications required by UltraSonix. Therefore, the AudioFile.so loadable module creates a "work crew" of threads to perform signal processing within the UltraSonix process itself. Filtering is done within UltraSonix, rather than when the NetAudio.so module is used and filtering is done within the server.

9.4.5.3 Caveats

To use a multithreaded ("MT") loadable module, UltraSonix itself must be recompiled with the -mt option. By default, the system as shipped is not compiled with this option. Therefore, recompilation is necessary to use AudioFile.so.

9.4.6 Alva

(NOTE: this section is not finished.)

9.4.7 Genovations

9.4.7.1 Using the Genovations Keypad with UltraSonix

The Genovations Keypad is an external serial device with a standard set of numeric keypad controls. This device can be used as a (limited) alternative to the "standard" workstation keyboard typically used to navigate in UltraSonix.

To use the Genovations device with UltraSonix starts, insert the following line into your config file:

keypadLoadable = "Genovations"

This will load the Genovations.so shared object file into the UltraSonix process.

By default, UltraSonix assumes that the Genovations keypad is attached to serial port labeled as "/dev/ttya". The user can set an environment variable, GENOVATIONSDEV, to override this setting.

Note that the current implementation of Genovations.so does not start a server process, so the keypad must be attatched directly to the local machine. Future versions may support remote access to the keypad.

Be sure that there are no programs, services, or alternate configurations that could potentially lock the serial port. One common problem is that Solaris by default assumes that the serial port is used to run a dumb terminal. Use the "admintool" to turn the terminal services off the serial port.

9.4.7.2 Implementation Details

The current "bindevent" mechanism used by UltraSonix does not allow the association of TCL procedures with user input other than the X event stream. Thus, there is no way via "bindevent" to associate a particular piece of TCL code with a key press on the Genovations keypad.

Instead, the Genovations.so code "directly" invokes certain actions whenever key sequences are pressed. These actions are "hard-wired" in the Genovations code and cannot be changed without recompiling.

The current mapping of keypad events to Actions is:

KEY	ACTION
0	StopSpeaking
1	ReadThisChar
2	DownPressed
3	ReadThisWord
4	LeftPressed
5	FivePressed
6	RightPressed
7	ReadThisSentence
8	UpPressed
9	ReadThisSentence
.	ChangeTextMode
enter	SelCurrent

10.0 MISCELLANEOUS TOPICS

This section describes a number of miscellaneous topics that may be of interest to developers maintaining the UltraSonix source code. This section discusses the subprocess management APIs available within the system, and the console subsystem.

10.1 Process Management

This section describes the facilities provided by UltraSonix for managing subprocesses. Its primary audience is programmers maintaining the internals of UltraSonix, or developers of loadable modules who need to spawn server processes to manage I/O.

10.1.1 Introduction to Process Management

Occasionally UltraSonix must spawn child processes to accomplish some task. The chief situations where subprocesses are started are the console application, and any device-specific I/O servers which may be required by your particular hardware configuration.

Subprocess management is accomplished via the ProcessManager class. There is only one instance of this class in UltraSonix, maintained by the global Mercator instance. To retrieve the ProcessManager instance in your code, do the following:

```
// Declare the single global Mercator instance.
extern Mercator *mercator;

// Get the ProcessManager.
ProcessManager *pm = mercator->GetProcessManager();
```

The ProcessManager class provides the following facilities:

- "Wrapper" functions to start, stop, and signal processes.
- "SIGCHLD handling for all subprocesses (to prevent "zombie" processes.)
- Optional automatic shutdown of subprocesses on exit.
- Subprocess tracking.

The goal of the ProcessManager class is to provide a single point of control for all subprocess-related operations to (1) prevent possible programmer errors, and (2) provide orderly control over subprocess termination.

10.1.2 Using the Process Manager: Basic

Most code will use only one method on the ProcessManager: StartProcess(). The StartProcess() method is used to spawn a new child process. Here is the declaration of the StartProcess() method:

```
pid_t   StartProcess(int argc, char * const argv[],
                     char *const envp[] = NULL,
                     int killOnShutdown = TRUE,
                     UnexpectedDeathProc unexpected = NULL,
                     BeforeDeathProc before = NULL,
                     AfterDeathProc after = NULL);
```

Note that the assignment operators in the declaration provide C++ "default parameters" to these arguments. If these parameters are omitted in the call to StartProcess() the default values will be used. So "simple" use of StartProcess only requires argc and argv.

The first four parameters (argc, argv, envp, killOnShutdown) are the most important for basic use and are described here. The other parameters are described under "Using the ProcessManager: Advanced."

argc	Argument count
argv	Argument vector
envp	Process environment
killOnShutdown	Should the process be terminated when UltraSonix is shutdown

The argc and argv parameters are required. ProcessManager() examines argv[0] to try to locate the executable filename. If argv[0] contains a slash character, argv[0] is treated as the actual filename. If it does not contain a slash, the user's PATH is searched for the specified filename. This behavior allows absolute pathnames or PATH searching to be employed.

Envp is optional. If present, it is used as the subprocess's environment; if omitted, the parent's environment is used. This feature is primarily used to allow UltraSonix to spawn subprocesses which are themselves UltraSonix clients (that is, they use the UltraSonix-modified X libraries to communicate properly with the UltraSonix system).

The killOnShutdown parameter is a boolean which specifies whether the ProcessManager will terminate the subprocess when UltraSonix itself shuts down. Child processes are terminated via a SIGTERM signal.

StartProcess() returns the process ID of the subprocess which was created, or -1 of an error occurred.

10.1.3 Using the Process Manager: Advanced

The ProcessManager allows callers to register three functions that can be used to handle special subprocess needs. The prototypes of these handler functions are as follows:

```
typedef int (*UnexpectedDeathProc)(pid_t);
typedef int (*BeforeDeathProc)(pid_t);
typedef int (*AfterDeathProc)(pid_t);
```

All of these arguments have default parameters of NULL.

The UnexpectedDeathProc, if supplied, will be called by ProcessManager when the child process terminates "unexpectedly" (that is, not as a result of a call to StopProcess() or the normal shutdown procedure). A common use of this procedure might be to restart an I/O daemon that has died. Note that if a daemon process is restarted, any code that was connected to that daemon will have to reconnect.

The BeforeDeathProc and AfterDeathProc arguments, if supplied, are called before and after (respectively) UltraSonix performs an "orderly" termination of the subprocess. These can be used for any specific clean-up code that might need to be run when a process is halted.

In addition to StartProcess(), the ProcessManager provides several other APIs:

int StopProcess(pid_t pid)

This method stops the process specified by pid via a SIGTERM signal. Even processes that are not started from ProcessManager may be stopped using this method.

int SuspendProcess(pid_t pid)
int ResumeProcess(pid_t pid)

These methods are used to suspend and resume processes (whether or not they were started from ProcessManager) via SIGSUSPEND and SIGRESUME signals.

int SignalProcess(pid_t pid, int signal)

SignalProcess() is used to send arbitrary signals to a process. Note that if you use SignalProcess() to send a SIGTERM or other fatal signal to a process, the death is considered unexpected (and hence, the UnexpectedDeathProc will be called if available). Killing a process via StopProcess() provides an orderly shutdown (and call of BeforeDeathProc and AfterDeathProc, if available).

ProcessManager::ProcessNode *FindProcess(pid_t pid)

FindProcess() is used to retrieve per-process information given the pid of a process. This call can only retrieve information for processes started via ProcessManager. See ProcessManager.h for the type definition of ProcessNode. Callers of this function can update information in a process's ProcessNode, but they are discouraged from doing so, as this information is used internally by ProcessManager. Callers should NEVER free the data returned by FindProcess(). The method will return NULL if the specified process has no associated ProcessNode.

10.1.4 Process Manager Implementation

ProcessManager is implemented as an FDInterest subclass. When an instance of the class is intialized, it creates a pipe for reading and writing, and installs a signal handler (called Reaper) to catch SIGCHLD signals.

The Reaper function, when it detects that a child process has died, writes the process ID and exit status of the child onto the writer end of the pipe.

The reader end of the pipe has been registered with the FDInterest superclass as an "interesting" descriptor. so it will become available for reading after Reaper has run. The HandleActivity() method will be dispatched, which collects the process ID and status of the child process.

After requisite error checking, HandleActivity() will do the following:

- Determine if the shutdown was expected. An "expected" shutdown is denoted by the "flaggedForKill" boolean being set to TRUE in the process's ProcessNode. This flag is set to TRUE whenever StopProcess() is called to terminate a process.
- If the death is unexpected, it calls the UnexpectedDeathProc for the procedure, if it exists.
- If the death is expected, it calls the AfterDeathProc for the procedure, if it exists.
- The ProcessNode for the process is removed from the dictionary of ProcessNodes.

The reason for the pipe implementation is to prevent arbitrary code form being run inside the signal handler. The signal handler is very simple: it merely has to catch the child exit status and write the process ID and status onto a pipe and return.

The more complicated processing of data structure manipulation and call of (perhaps arbitrary) client-supplied death procedures is relegated to HandleActivity().

10.2 The Console

10.2.1 Using the UltraSonix Console

UltraSonix Configuration (/opt/GTsonicx/bin/gui-console) is a GUI based console which allows the user to configure some of the attributes of UltraSonix interactively. The users can also issue commands in the window from which UltraSonix Configuration is started. (Eventually this will be incorporated into a text area in the console itself.) The utility currently allows the user to set the speech rate, speech voice and the user level with toggle buttons.

10.2.2 Starting a Console

UltraSonix can start the console automatically as a child process. When UltraSonix starts it looks for the string list variable "console" in the config file (default is /opt/GTsonicx/etc/mercator.config). The variable should contain a list of strings which form the command to start the console. The first string in the list should be the absolute path and name of the console program, and the rest of the list should contain any command line options to be passed to the console.

The variable, "consoleWait", in the config file specifies how long UltraSonix will try to connect to the console via named pipes. If the connection is not established after this time then UltraSonix assumes that the console fails to start and it will revert back to stdin and stdout for input and output.

10.2.3 Console Environment

The GUI based console in the package requires a different LD_LIBRARY_PATH environment variable then UltraSonix. This may be true for other consoles as well. To accommodate this, UltraSonix looks for another string list variable "consoleEnv". Each string in this variable should be in the form:

"VAR=VALUE"

where VAR is the name of the environment variable and VALUE is its value when the console is running. For example, an Motif based console applications would need to have its LD_LIBRARY_PATH set to include paths to the modified R5 and RAP libraries. The value of LD_LIBRARY_PATH from the user's shell environment will be replaced with the one specified in consoleEnv. (The user can also unset an environment variable by including the variable's name in the consoleEnv string list without any value.)

10.2.4 Example

```
###
### Set which program to run as console
###
###
console         = ("/opt/GTsonicx/bin/gui-console")
consoleWait     = 5
consoleEnv      =
("LD_LIBRARY_PATH=/opt/GTsonicx/lib/RAP:/opt/GTsonicx/lib/R5:/opt/X11R5/lib:/net/hm2/p
ackages/X11R6/lib:/usr/dt/lib")
```

10.2.5 Implementation Details

The UltraSonix Configuration Utility creates a pair of named pipes: /tmp/from-console and /tmp/to-console, when it first starts up. These will be opened by UltraSonix if they have been created when it starts up. Using these named pipes the Configuration utility is able to issue Tcl commands that UltraSonix already understands directly, as if they were typed in by a user. The GUI part of the code then simply sets up appropriate callback functions to issue these commands depending on which button is pushed. To allow the user type in Tcl commands in addition to selecting buttons from the interface and to print out feedback from UltraSonix, we also added callback functions to monitor the standard input and the pipe /tmp/to-console and added callbacks to handle them. (see mconfig.c) When input is detected from stdin, the callback function readInput simply copies a line to a buffer and then send it to UltraSonix via the /tmp/from-console pipe. When any activity is detected on the /tmp/to-console pipe, the callback function printMercatorOutput is called to dump outputs from UltraSonix to standard output.

Another important issue when implementing an GUI console is that the console is usually started before or at the same time as UltraSonix. This means that UltraSonix will not be able to detect its existence right away. To solve this problem the GUI console must issue a connect command explicitly to UltraSonix, with its own window id, so that UltraSonix will also be able to navigate the console. The connect command is issued to UltraSonix via the /tmp/from-console pipe, as other console commands, and it should be issued after the pipes have been created and also after the GUI have been managed or popped up to the desktop in order to get its window id.

11.0 Appendix: TCL Command Reference

(Note that words which are not enclosed in brackets '<' and '>' are keywords. Words enclosed in brackets are variables. Words which are enclosed by square brackets '[' and ']' are optional.)

11.1 TCL Interfaces to C++ Methods

This section contains built-in TCL commands which are implemented in the files Interp.h and Interp.cc. These commands generally parse the expected TCL syntax and then call the corresponding C++ methods.

11.1.1 Diagnostic Output

The Error command can be used to display an error message, get the current error message level and set the error message level. To display an error message:

11.1.1.1 Displaying Error Messages

Error <error level> <proc name> [<msg 1> <msg 2> <msg 3> ...]

<error level> sets the error level of the error messages in the call. If it is less than or equal to the current error level, then the messages will be displayed. Possible values for <error level>, in increasing order, are:

EL_ABORT	(abort after displaying the error messages)
EL_FATAL	(exit after displaying the error messages)
EL_ERROR	(messages are the result of an error condition)
EL_WARNING	(warning messages)
EL_STATUS	(status messages)
EL_INFORMATIONAL	(informational messages)
EL_DEBUG	(debugging messages)

<proc name> is the name of the procedure which contains the Error call. <msg n> are the actual error messages.

11.1.1.2 Getting and Setting Error Levels

Error level [<err level>]

UltraSonix maintains a current error level, which determines which error messages are displayed. All messages with error levels equal to or less than the current error level will be displayed. In addition to the seven possible values for error level, there are two additional values which can be set:

EL_NO_MESSAGES	(do not display any debugging or error messages)
EL_ALL_MESSAGES	(any messages)

11.1.2 Operations on Clients

This section describes TCL operations that can be used to determin the currently active client and switch among clients.

11.1.2.1 Determining the Current Client

currentClient

The currentClient command returns a unique identifier for the current client (which can be used in calls to other TCL commands that support client operations), or the string "NULL" if there is no current client.

11.1.2.2 Moving Between Clients

advanceClient
backupClient

These two commands cycle through the list of currently active clients. They have no return value.

11.1.2.3 Client Names

client name <client>

The client command is used to retrieve information about specified clients. Currently only the "name" option is supported, which returns the name of the client as specified by the application writer.

11.1.3 Operations on Objects

11.1.3.1 Determining the Current Object

currentObject

The currentObject command returns the name of the current object if there is one, or the string "NULL" if there is no current object.

11.1.3.2 Converting Object Names

long2short <obj>
short2long <obj>

These two commands are used to convert between the "long" (Xrm-style, dot-notation) and "short" (unique object identifier) names of XtObjects. They each return the complementary name.

11.1.4 Binding Events and Actions

11.1.4.1 Associating TCL Procedures with Events

bindevent [<obj|client>] press|release [<shift|ctl|meta|alt>] <key> <proc>

The bindevent command is used to associate a named TCL procedure with a particular key event. The procedure will be executed automatically whenever the event occurs within the scope specified by the arguments to bindevent.

The first argument to bindevent is optional, and can be an identifier for either an XtObject or a client. If the argument is present, it indicates that the bind should be established only on the particular object or client specified. If the argument is not present, the bind will be established globally (that is, on all objects in all clients).

The next argument indicates whether the TCL procedure will be invoked on a key press or a key release. The next argument indicates zero or more (optional) modifier keys to detect. Next, the actual keysym is indicated (see /usr/openwin/include/X11/keysymdef.h for a list of keysyms). Finally, the TCL procedure is named. This procedure will be executed whenever the indicated keysym is either pressed or released, with the optional set of modifiers enabled, within the scope indicated by the optional first argument.

11.1.4.2 Actions

addaction <proc>

The addaction command makes the named TCL procedure "visible" to the C++ action interface used internally by UltraSonix. At key points, UltraSonix will "call out" to named actions when the state of the interface model changes. To enable this callout, the addaction command must be used.

callaction <action> <arg0> <arg1> <...>

The callaction command uses the C++ Action interface to invoke the named action with the specified arguments. It provides a mechanism for TCL code to invoke the same actions as C++ code.

11.1.5 Using the Braille Terminal

11.1.5.1 Sending Text to the Braille Device

braille display "<row 1>" ["<row 2>" "<row3>" ...]

Sets the text buffer of the braille loadable object to <row 1> <row 2> <row 3>.

braille status <text>

Sets the status cells of the braille device to <text>.

11.1.5.2 Jump Scroll Mode

braille jumpScroll <on/off>

Turns the jump scroll mode on or off.

11.1.5.3 Setting Braille Translation Table

braille translate <table>

Sets the braille translation table to <table>.

11.1.5.4 Querying Braille Device Capabilities

braille cap <capability>

The following capabilities can be queried with this call:

displayCells	(returns the number of display cells)
statusCells	(returns the number of status cells)
highlightSupported	(returns 1 if highlighting is supported, 0 otherwise)
hasCursorKeys	(returns 1 if the device has cursor keys, 0 otherwise)
hasProgKey	(returns 1 if the device has program keys, 0 otherwise)
hasHomeKey	(returns 1 if the device has home keys, 0 otherwise)
otherKeys	(returns 1 if the device has other key, 0 otherwise)

11.1.6 Console Operations

connectPipe
disconnectPipe

These two commands are used to either establish a connection to the console over a named pipe (connectPipe), or break an existing connection to the console over a named pipe (disconnectPipe).

When issuing the connectPipe command, the console should already be running and blocked, waiting on UltraSonix to attempt to connect to it.

If UltraSonix disconnects from its console, it will revert to standard input/output for debugging messages.

11.1.7 Connecting to Clients

connect <winid>

The connect command is used to issue an explicit request for UltraSonix to initiate the connection procedure on a specified window ID.

11.1.8 Key and Button Events

11.1.8.1 Using Keyboard Identification Mode

key announce [onloff]

This command enables or disables keyboard identification mode. While in keyboard identification mode, all keyboard input is spoken directly, and is not processed by UltraSonix.

The key combination Shift-Ctrl-Q will also terminate keyboard identification mode.

11.1.8.2 Generating Keyboard Input to Applications

key <object> <key> [<shift|control|meta|alt>] <press|release|both> <stat>

This command is used to send synthetic keyboard events to an application. The events will be sent to the window associated with the object specified by the <object> parameter. The <key> parameter indicates the basic key to send; it is espressed as an X keysym (the string "Right" is used to indicate the Right arrow, for example. See /usr/openwin/include/X11/keysymdef.h for a list of keysyms.) Next, an optional modifier parameter allows the caller to specify a modifier key, such as shift, control, and so on.

The key command allows callers to generate presses and releases individually or, more commonly, both sequentially. The next parameter specifies whether to send a press, a release, or both.

The final parameter, <stat>, indicates the current grab status of the key on the current object. UltraSonix "grabs" the keys it needs for navigation away from applications. The keys that are grabbed are determined by the "bindevent" commands present in the startup TCL files. UltraSonix does not "remember" what keys are already grabbed; this is left up to the writer of TCL code. You must pass either "1" or "0" as the <stat> parameter to indicate whether the key you are generating is one that has already been grabbed or not.

If you are sending a key that is grabbed by UltraSonix, the key command will ungrab it, send the required events, and then grab it again. If you send a key that has been grabbed without setting the <stat> argument to "1" then UltraSonix may loop indefinately (the key will be sent to the application, but the grab is still active so it is returned to UltraSonix, which is again passed to the application, ...).

11.1.8.3 Generating Mouse Input to Applications

11.1.8.3.1 Button Events

button <object> [press|release] [1|2|3]

The button command generates button presses and releases to applications. The command allows callers to specify whether a press or release is generated, and which button to generate (1, 2, or 3).

The optional "object" parameter indicates an explicit object to send the input to. If present, the mouse will be warped to the indicated object and input will be sent there. If not present, the event will be sent to the object currently under the pointer.

11.1.8.3.2 Moving the Cursor

relativemotion <x> <y>

The relativemotion command is used to move the on-screen cursor to a new location. The x and y arguments indicate the offset to the new position, relative to the current position. Also note the "xwin warp" command.

11.1.9 Retrieving Properties of the Model

The "modelManager" command is used to retrieve information from the off-screen model. There are several options available to this command.

11.1.9.1 Parent/Child Relationships

modelManager parent <object>

The parent option returns the identifier for the parent of the specified object, or the string "NULL" if the object has no parent.

modelManager children <object>

The children option returns a TCL list of all of the children of the current object, or an empty list if the object has no children.

11.1.9.2 Object Location and Geometry

modelManager window <object>

The window option returns the actual X Window System window identifier for the window associated with the specified object. Two special values may be returned: 0 indicates that the object is not realized (that is, it has no window associated with it), and 2 indicates that the object is a gadget (that is, it is a windowless widget).

modelManager x <object>
modelManager y <object>
modelManager width <object>
modelManager height <object>
modelManager borderWidth <object>

These options are used to return the X and Y location, width and height, and border width of the specified object. The return values are in terms of pixels. X and Y are relative to the object's parent.

modelManager location <object>

The location object issues an explicit RAP request to retrieve the specified object's location. In general this option should never be used, as it incurs a significant performance penalty. It may be useful in some situations where the off-screen model is out-of-date with respect to the on-screen display, however.

11.1.9.3 Names and Other Object Attributes

modelManager name <object>

The name option returns the name of the specified object, as given by the application writer.

modelManager class <object>

The class option returns the widget class of the specified object.

modelManager longname <object>

The longname option returns an "Xrm-style" dot-notation name for the specified object. The value returned from this command may be used in a .Xdefaults file, or as an identifier in an object template to refer to a specific widget.

modelManager mapped <object>
modelManager managed <object>

These options return the string "TRUE" or "FALSE" depending on whether the specified object is mapped or managed, respectively.

11.1.10 Generating Non-speech Audio Output

playsound <sound file> [<rate> <volume> <muffle> <looped> <delay>]

Plays <sound file>. If more than two arguments are supplied, it also sets the playing rate (0 to 1.0), volume (0 to 100), muffle (0 to 100?), looped (0 or 1) and delay (in milliseconds).

stopAllSounds

The stopAllSounds command stops all sounds currently playing on the audio device.

audio name

The audio command returns the name of the current audio device (for example, "netaudio").

11.1.11 Shutting Down UltraSonix

quit

The quit command is used to terminate UltraSonix gracefully.

11.1.12 Accessing Resources

resource <obj> get <name>

The get option to the resource command retrieves the value of the named resource, in the named object, from the off-screen model. It is presented as a string.

resource <obj> set <name> <value>

The set option to the resource command changes the value of the resource WITHIN THE APPLICATION. This option should be used with extreme caution, as it is possible to break applications with this command. It is also very expensive.

To work, a converter from string type to the native type of the resource must exist within the application.

resource <obj> request <name>

The request option to the resource command issues an explicit RAP request to retrieve the value of the specified resource. This option should almost never be used, as it is very expensive. It may be useful in situations where application- or widget-writers update the value of resources in a way that bypasses the RAP hooks in X.

resource <obj> list

The list option lists all of the resources, their types, and their values on the specified object.

11.1.13 Speech Output

11.1.13.1 Producing Speech

speak <text> [nointerupt]

This command sends <text> to the speech device. If the nointerupt keyword is not given, any current speech will be stopped before <text> is spoken. If the keyword 'nointerupt' is given, then <text> will be appended after any current speech.

11.1.13.1 Querying Speech Device Capabilities

speak cap <capability>

This command queries UltraSonix for specific capabilities of the speech device. Valid values for <capability> are:

voices (returns a list of voices supported)
languages (returns a list of languages supported)
minRate (returns the minimum speech rate)
maxRate (returns the maximum speech rate)
gainSupported (returns 1 if gain control is supported by the device,
0 otherwise)

11.1.14 Low-Level X Window Operations

This section describes the usage of the "xwin" command, and its associated options, to operate on low-level X windows.

11.1.14.1 Using Properties

xwin property set <winid> <property> <value>
xwin property get <winid> <property>

The property option of the xwin command allows the user to set and get the values of window properties. THIS OPTION IS NOT FULLY IMPLEMENTED.

11.1.14.2 Using Selections

xwin selection set <selection> <value>
xwin selection get <selection>

The selection option of the xwin command allows the user to set and get named X selections (for cut and paste). THIS OPTION IS NOT FULLY IMPLEMENTED.

11.1.14.3 Accessing Window Attributes

xwin window set <winid> <attribute> <value>
xwin window get <winid> <attribute>

The window command allows the caller to retrieve named window attributes from specified windows, and to set those attributes. THIS OPTION IS NOT FULLY IMPLEMENTED.

11.1.14.4 Window and Pointer Management

xwin warp <x> <y>
xwin warp <obj>

The warp option to the xwin command allows the caller to "warp" the mouse pointer to a specified location on the screen. There are two forms of this command. The first specifies an absolute X,Y coordinate pair (relative to the root, or background, window). The second specifies a specific object to warp to. The cursor will be placed at coordinates (1,1) relative to the new object.

xwin raise <obj>
xwin lower <obj>

The raise and lower options are used to raise or lower the window hierarchy containing the specified object.

setFocus <obj>

The setFocus command sets the current location and window focus to the specified object, and sets the current client to the client that contains this object.

NOTE that setFocus should be a suboption of the xwin command for orthogonality.

11.1.15 Using the ScreenReader

ScreenReader functionality is accessed via the "sreader" command. There are a number of options to this command that can be used to change ScreenReader parameters, retrieve text from the model, and query and update cursor positions.

11.1.15.1 Changing ScreenReader Parameters

sreader speed <speed>
sreader voice <voice>

The speed and voice options globally change the current speech rate and voice. Note that these options affect the way UltraSonix speaks in every text area.

11.1.15.2 Using Cursors

sreader <obj> togglecursor

The togglecursor option toggles between edit and review mode in the specified object.

sreader <obj> togglefollow

The togglefollow option toggles whether the review cursor will follow the edit cursor as it moves in the specified object.

sreader <obj> cursor

The cursor option returns the current cursor mode (either edit or review), the current follow mode (either review-follows-edit, or not), and the current X, Y cursor position for the specified object.

sreader <obj> cursor mode

The cursor mode option returns the cursor mode (either review or edit) for the specified object.

sreader <obj> cursor coords

The cursor coords option returns the current cursor location.

sreader <obj> cursor <x> <y>

The cursor option, with X and Y arguments, sets the position of the current cursor in the specified object to the provided location.

11.1.15.3 Using Filters

sreader <obj> filter

The filter option, with no arguments, returns the number of installed filters in the specified object.

sreader <obj> filter <filter>

The filter option provided with the number of a filter returns the status of the filter in the specified object (either ON or OFF).

sreader <obj> filter <filter> <status>

The filter option used with the number of a filter and a status (either 1 or 0) sets the filter either on or off, for the specified object.

sreader <obj> filter <filter> name

The filter name option returns the ASCII name of the filter whose ID is provided by the <filter> argument.

11.1.15.4 Reading and Moving Through Text

sreader <obj> read <this|next|prev> <char|word|line|sentence|paragraph|screen> [raw]

The read option returns the specified unit of text to the TCL interpreter. It does not advance the cursor position. The text is returned from whichever cursor is currently active (either review or edit).

The first argument specifies the object from which text should be returned. The argument after "read" indicates whether the text chunk that is returned should be at the current cursor ("this"), before the current cursor ("prev"), or after the current cursor ("next").

The next argument specifies the size of the text chunk to be retrieved. Text may be retrieved by character, by word (words are separated by one of a set of delimiter characters), by sentence (sentences are separated by one of a set of terminating characters), paragraphs (separated by blank lines), lines, or screens (all currently visible text). The optional "raw" parameter indicates that the text should be returned without filtering being applied to it. This option is useful for retrieving text that will be sent to a braille device.

(Note that terminal and delimiter characters may be set via the C++ API to the ScreenReader class. There is currently no TCL API for setting these characters.)

sreader <obj> move <this|next|prev> <char|word|line|sentence|paragraph|screen>

The move option works the same as the read option, with the exception that it updates the position of the currently active cursor. Move returns the text is has moved over. (Note that the move option does not currently support the "raw" option.)

11.1.15.5 Miscellaneous ScreenReader Functions

sreader <obj> incoming

The incoming option cycles through the "incoming text" modes (also known as echo modes). Supported echo modes are:

character	Speak incoming text as each character is typed.
word	Speak incoming text as each word is completed.
line	Speak incoming text as each line is completed.
sentence	Speak incoming text as each sentence is completed. (Not currently supported.)
click	Click as each character is typed. (Not currently supported.)

sreader <obj> dumptext

Primarily useful for debugging, the dumptext option displays all text currently in the ScreenReader's text model for the specified object.

11.1.16 Miscellaneous Text-Related Functions

11.1.16.2 Determining the Location of Text

getTextVertCoord <obj>
getTextHorizCoord <obj>

These two commands return the pixel location of the current cursor in the text area specified by the object parameter. The location is expressed as a relative offset within the text area.

11.1.16.1 Debugging the Text Model

textrepdebug <obj> [on|off]

The textrepdebug command provides a visual debugging tool for the text model. To enable the TextRep debugging window for a particular object, pass the name of the object and the "on" parameter to textrepdebug. If the object supports text mode, a new window will be created which provides a representation of what UltraSonix "thinks" is in the text area. Issue the "off" parameter to disable text rep debugging.

11.1.17 Logging User Activities

UltraSonix provides a number of commands to support logging and tracing of interactive sessions. These commands can be used to evaluate user performance when using the system.

StartTimer <file>

The StartTimer command opens the specified file and writes the current time of day information into it. It is assumed that this file will later be used to collect timed performance data via the logtoFile command.

logtoFile <file> <arg0> <arg1> <...>

The logtoFile command writes the current time and all of its command line arguments to the file named by the first argument.

11.1.18 Template and Configuration Management

This section describes the commands used to interact with the template and configuration subsystems in UltraSonix.

11.1.18.1 Using Template Values From TCL

wtemplate <obj> <attribute>

The wtemplate command retrieves template attributes from the "widget" templates (class and object templates, as opposed to application templates). The user supplies the name of an object, and an attribute to retrieve. The command will return the value of the attribute, searching in both object and class template lists, and running any TCL procedures necessary.

NOTE that there is no corresponding atemplate (for app template) command (although there should be).

11.1.18.2 Writing New Template Files

writeout <templatename> <filename> [<A|C|O>]

The writeout command is used to store the internal (C++ object) representation of a template to long-term storage. The result is a new template file, which is parsable and loadable by UltraSonix.

The first argument is the name of the template. This name should be appropriate for the type of template file you are writing (dot-notation for object templates, class name for class templates, application name for app templates). The next argument is the filename to write the template to.

The final (optional) argument is a specifier indicating whether the template is an application, class, or object template. If omitted, the writeout command will search for an existing template with the specified name, determine its type, and use that type when writing the template. We recommend always specifying the template type explicitly.

11.1.18.3 Retrieving Configuration Attributes

configVal <attr>

The configVal command retrieves attributes from the configuration file. This command is currently quite broken, and will only return the errorLevel attribute.

11.1.18.4 Loading Files

loadattrib
loadconfig
loadtemplates
loadall

The loadattrib, loadconfig, and loadtemplates commands cause UltraSonix to reload its attribute, configuration, and template information from the locations it was loaded from previously.

The loadall command forces a reload of all startup files.

11.2 "Pure" TCL Commands

These are procedures which are implemented in TCL and are thus interpreted at run time. Only the most important of these are described here; see the TCL files for more details.

11.2.1 Audio

Audio <node> <delay>

The Audio procedure (defined in audio.tcl) is responsible for playing the audio clip associated with a given object (specified by node), at a given delay. This procedure will retrieve any necessary widget-specific information to generate the correct sound.

AudioOn
AudioOff

These procedures will mute or unmute the non-speech audio from UltraSonix (mute state is controlled by a TCL global variable).

11.2.2 Interface Helpers

Select <node>

The Select procedure generates a mouse press-release pair on the object specified by the node argument.

Novice
Intermediate
Expert

These procedures set the user level to either novice, intermediate, or expert. The notion of "user levels" is only maintained by the TCL code as a global variable that is checked by the various other TCL procedures. The C++ side of UltraSonix knows nothing about user levels.

Preview

Produce an audio preview of the current object, if it is a container.

Info

Produce the informational non-speech and speech output for the current object.

HearPath

Produce a non-speech audio cue of the path from the top of the widget tree to the current location.

JumpBack
JumpForward

These procedures jump to the first or last object in a container.

11.2.3 Navigation

GoTo <node>

Move immediately to the specified node.

NavUp
NavDown
NavLeft
NavRight

These procedures are responsible for the basic inter-object navigation used by UltraSonix. They make an attempt to discard extraneous levels in the hierarchy.

NavAppRight
NavAppLeft

Move between applications.

11.2.4 Text Mode

ToggleCursorMode
ToggleFollowMode
ToggleFilterMode
FilterModeOn
FilterModeOff
AnnounceFilterMode
ChangeTextMode

These procedures are responsible for calling into the C++ layer to implement various cursor and filtering functions.

11.2.5 Miscellaneous

widgetTree <outfile>

The widgeTree procedure dumps the entire widget hierarchy containing the current object to the specified file.

12.0 Appendix: RAP Protocol Specification

See the following World Wide Web page for the RAP specification:
http://www.cc.gatech.edu/gvu/multimedia/x-agent/RAPBack.html

13.0 Appendix: Known Bugs

- There is sometimes a delay (buffering problem?) when popups are created. The user must navigate into the object before sounds are flushed.
- UltraSonix does not rendezvous with clients created with UIM/X. Problem is with how UIM/X creates its toplevel shells.
- Scrollbars are not used.
- When UltraSonix detects a new window being mapped on the root window, it should raise the current client to ensure that focus is not lost.
- Use of OpenWindows in left-handed mode breaks synthetic mouse input. UltraSonix should detect left-handed mode and generate events accordingly.

- If the last object in a client is unmapped or destroyed (but the client remains running) UltraSonix should select a new client to move to. It currently does not. (But we've never actually seen any applications that destroy all their widgets but remain active.)
- ProtoTextRep cannot support empty text areas well, because of it's dependence on detection of interline spacing.
- The client-side converter CvtXmStringTableToString() doesn't work. Hence we cannot download XmStringTable resources into UltraSonix.
- There is inconsistent usage of CheckIsShell and IsTopLevel in RAP.cc.
- NumChildrenMuffle in templateprocs.tcl does not check for divide by zero (breaks on containers with no children).
- UltraSonix has no way to block waiting on a RAP request to return. Hence, things like "conservative" retrievals of resource values that may be out-of-sync are impossible.
- We're still just sending button presses for selection. We need to get the translation table out of the widgets and send the *appropriate* events for selection instead.
- Right now we can only have one action bound to a particular name. It may be useful to support multiple actions, possibly from different Action subclasses, associated with a name.
- The bindevent command only understands X protocol events, not events from other input sources (such as a braille keyboard, external keypad, or speech recognizer). This causes some hackery in the loadable modules that support input.
- UltraSonix should support a "collaborative" mode where a sighted user can use the mouse to change UltraSonix' idea of where the current context is.
- The console should include a command line that accepts regular Unix commands, and executes them with an LD_LIBRARY_PATH appropriate for starting under UltraSonix.
- Iconified applications are not handled well.
- Copy/paste, and drag/drop are not supported.
- Resource converters (for the ResourceTypeManager) should by dynamically loadable so that they can be created without needing to recompile UltraSonix.
- An interactive customization/keybinding/template-editing facility would be nice.
- Text areas modeled via ProtoTextRep may not be cleared appropriately.
- If UltraSonix generates a very large amount of debugging output to the console, the console may die unexpectedly.
- Repeatedly hitting the buttons on the console very quickly may hang the console.
- Blanks added as "padding" to the text model (spaces at the ends of lines and blank lines not explicitly drawn by the application) may have different graphics context information than surrounding text.

- Detection of highlight changes across line boundaries does not work.
- Detection of highlighting implemented via rectangle drawing (X_PolyFillRectangle) is not supported.
- 16-bit text is not supported.
- Menus are only supported via tear-offs.
- ProtoTextRep does not support multiple fonts with different metrics in the same text area (multiple fonts with the *same* metrics are supported however).
- UltraSonix does not currently take advantage of information about spatial geometry of interfaces. Hence, it does not understand overlapped widgets (as in dtcm) or widgets that are invisible because they are clipped by their parents (as in dtfile).
- Horizontal scrolling in text areas is not handled well by ProtoTextRep.
- On keyboards with multiple sets of cursor keys (such as the Sun Type 5) navigation in edit mode on applications without block cursor support may be off by one.
- There are still occasional menu-related problems: sometimes UltraSonix will attempt to warp to an unposted menu (the symptom is that the pointer goes to location (0,0) on the screen).